Certified Tester

Advanced Level Syllabus

Technical Test Analyst (TTA)

Versão 2012br

Comissão Internacional para Qualificação de Teste de Software

---

Responsável pela tradução: BSTQB/TAG01 – Documentação

Baseada na versão 2012 do *CTAL Syllabus* do ISTQB.

Brazilian Software Testing Qualifications Board

- Subgrupo de trabalho de analistas de testes de nível avançado: Judy McKay (presidenta), Mike Smith, Erik Van Veenendaal, 2010-2012.

Histórico de revisões

Versão	*Data*	*Observações*
ISEB v1.1	04/09/2001	*ISEB Practitioner Syllabus*.
ISTQB 1.2E	Setembro de 2003	*ISTQB Advanced Level Syllabus* do Grupo de *Software* da Organização Europeia para a Qualidade (EOQ-SG, na sigla em inglês).
V2007	12/10/2007	Versão de 2007 do *Certified Tester Advanced Level Syllabus*.
D100626	26/06/2010	Incorporação das alterações aceitas em 2009 e divisão de capítulos por módulos diferentes.
D101227	27/12/2010	Aceitação de alterações no formato e correções que não afetam o significado das frases.
Esboço V1	17/09/2011	Primeira versão do novo *syllabus* para TTAs com base em um documento acordado de definição de abrangência. Revisão do grupo de trabalho de nível avançado.
Esboço V2	20/11/2011	Revisão da versão da comissão nacional.
Alfa 2012	09/03/12	Incorporação de todos os comentários das comissões nacionais recebidos em função do lançamento de outubro.
Beta 2012	07/04/2012	Envio da versão beta à Assembleia Geral.
Beta 2012	08/06/2012	Envio da versão revisada às comissões nacionais.
Beta 2012	27/06/2012	Incorporação de comentários do EWG e glossário.
RC 2012	15/08/2012	Lançamento de rascunho – inclusão de edições finais das comissões nacionais.
RC 2012	02/09/2012	Incorporação de comentários da BNLTB e de Stuart Reid. Cruzamento com Paul Jorgensen.
GA 2012	19/10/2012	Edições e correções finais para lançamento por parte da Assembleia Geral.

Índice

Agradecimentos

O presente documento foi elaborado pelo grupo principal do subgrupo de trabalho de nível avançado da International Software Testing Qualifications Board – *Technical Test Analyst* composto por: Graham Bath (presidente), Paul Jorgensen e Jamie Mitchell.

O grupo principal gostaria de agradecer à equipe de revisão e às comissões nacionais por suas sugestões e contribuições.

Quando o *Advanced Level Syllabus* foi concluído, o grupo de trabalho de nível avançado contava com os seguintes integrantes (em ordem alfabética):

Graham Bath, Rex Black, Maria Clara Choucair, Debra Friedenberg, Bernard Homès (vice-presidente), Paul Jorgensen, Judy McKay, Jamie Mitchell, Thomas Mueller, Klaus Olsen, Kenji Onishi, Meile Posthuma, Eric Riou du Cosquer, Jan Sabak, Hans Schaefer, Mike Smith (presidente), Geoff Thompson, Erik van Veenendal e Tsuyoshi Yumoto.

As seguintes pessoas revisaram, comentaram e escolheram este *syllabus*:

Dani Almog, Graham Bath, Franz Dijkman, Erwin Engelsma, Mats Grindal, Dr. Suhaimi Ibrahim, Skule Johansen, Paul Jorgensen, Kari Kakkonen, Eli Margolin, Rik Marselis, Judy McKay, Jamie Mitchell, Reto Mueller, Thomas Müller, Ingvar Nordstrom, Raluca Popescu, Meile Posthuma, Michael Stahl, Chris van Bael, Erik van Veenendaal, Rahul Verma, Paul Weymouth, Hans Weiberg, Wenqiang Zheng e Shaomin Zhu.

A Assembleia Geral da ISTQB® lançou este documento formalmente no dia 19 de outubro de 2012.

0 Introdução ao syllabus

0.1 Objeto do documento

O presente *syllabus* forma a base para a qualificação internacional de testes de *software* no nível avançado para o *Technical Test Analyst*. A ISTQB® fornece o *syllabus*:

1. Às comissões nacionais a fim de traduzi-lo para o idioma local e com a finalidade de credenciar os provedores de treinamento. As comissões nacionais podem adaptar o *syllabus* às suas necessidades linguísticas específicas e modificar as referências para adaptá-las às publicações locais;
2. Às comissões avaliadoras para a elaboração de perguntas no idioma local adaptadas aos objetivos de aprendizagem de cada *syllabus*;
3. Aos provedores de treinamento para produzirem os materiais dos cursos e definirem os métodos adequados de ensino;
4. Aos candidatos à certificação para prepará-los para a avaliação (em função de um curso de treinamento ou independentemente disso);
5. À comunidade internacional de engenharia de *software* e sistemas para fomentar o desenvolvimento da profissão de teste de *software* e sistemas e servir de base para livros e artigos.

A ISTQB® pode permitir que outras entidades utilizem este *syllabus* para outras finalidades, contanto que procurem e obtenham uma autorização prévia por escrito.

0.2 Panorama

O nível avançado é composto por três *syllabi* diferentes:

- *Test Manager*;
- *Test Analyst*;
- *Technical Test Analyst*.

O documento que faz um panorama do nível avançado [ISTQB_AL_OVIEW] contém as seguintes informações:

- Os resultados de negócios de cada *syllabus*;
- Um resumo de cada *syllabus*;
- Os relacionamentos entre os *syllabi*;
- Uma descrição dos níveis cognitivos (níveis K);
- Os anexos.

0.3 Objetivos de aprendizagem sujeitos à avaliação

Os objetivos de aprendizagem fundamentam os resultados comerciais e são utilizados na criação de avaliações para a obtenção da certificação *Advanced Technical Test Analyst Certification*. Em geral, todas as partes deste *syllabus* estão sujeitas a uma avaliação no nível K1. Isto é, o candidato reconhecerá e se lembrará de um termo ou de um conceito. Os objetivos de aprendizagem nos níveis K2, K3 e K4 aparecem no começo do capítulo pertinente.

0.4 Expectativas

Alguns dos objetivos de aprendizagem do *Technical Test Analyst* pressupõem experiência básica nas seguintes áreas:

- Noções gerais de programação;
- Noções gerais de arquitetura de sistemas.

1 As tarefas do Technical Test Analyst nos testes baseados em riscos – 30 minutos

Palavras-chave

risco de produto, análise de riscos, avaliação de riscos, identificação de riscos, nível de risco, mitigação de risco, teste baseado em risco

Objetivos de aprendizagem das tarefas do Technical Test Analyst nos testes baseados em riscos

1.2 Avaliação de riscos

TTA-1.3.1 (K2) Resumir os fatores genéricos de risco que o *Technical Test Analyst* normalmente precisa levar em consideração.

1.3 Objetivos de aprendizagem comuns

O seguinte objetivo de aprendizagem diz respeito ao conteúdo discutido em mais de uma seção deste capítulo.

TTA-1.x.1 (K2) Resumir as atividades de planejamento e execução de testes do *Technical Test Analyst* nos termos de uma abordagem baseada em riscos.

1.1 Introdução

No geral, o *Test Manager* tem a responsabilidade de definir e gerenciar uma estratégia de testes baseados em riscos. Normalmente, o *Test Manager* solicita o envolvimento do *Technical Test Analyst* para garantir que a abordagem baseada em riscos seja implementada corretamente.

Por conta de sua *expertise* técnica específica, o *Technical Test Analyst* participa ativamente das seguintes tarefas de testes baseados em riscos:

- Identificação de riscos;
- Avaliação de riscos;
- Mitigação de riscos.

Estas tarefas são realizadas iterativamente ao longo do projeto para lidar com os riscos de produto que surgirem e as mudanças de prioridades e para avaliar e comunicar as situações de risco com regularidade.

O *Technical Test Analyst* trabalha com a estrutura de testes baseados em riscos definida pelo *Test Manager* para o projeto. Ele compartilha o conhecimento que tem sobre os riscos técnicos inerentes ao projeto, como riscos relacionados à segurança, à confiabilidade e ao desempenho do sistema.

1.2 Identificação de riscos

Ao recorrer ao maior número possível de *stakeholders*, o processo de identificação de riscos provavelmente detectará o maior número possível de riscos significativos. Como o *Technical Test Analyst* possui habilidades técnicas singulares, é particularmente apto para a realização de entrevistas com especialistas, de *brainstorming* com colegas e de análises de experiências atuais e anteriores para saber onde estão as prováveis áreas de risco de produto. Em particular, o *Technical Test Analyst* deve colaborar com seus colegas técnicos (por exemplo, desenvolvedores, arquitetos, engenheiros de operações) para determinar as áreas de risco técnico.

Entre os exemplos de riscos que podem ser identificados estão:

- Riscos de desempenho (por exemplo, a incapacidade de atingir certos tempos de resposta sob condições de alta carga);
- Riscos de segurança (por exemplo, a divulgação de dados sensíveis através de ataques contra a segurança do sistema);
- Riscos de confiabilidade (por exemplo, a incapacidade de cumprir a disponibilidade estipulada no acordo de nível de serviço).

As áreas de risco referentes às características de qualidade específicas do *software* são discutidas nos capítulos relevantes deste *syllabus*.

1.3 Avaliação de riscos

Embora a identificação de riscos procure identificar o maior número possível de riscos pertinentes, a avaliação de riscos é o estudo dos riscos identificados para a categorização de cada risco e a determinação da probabilidade e do impacto relacionado a cada risco.

A determinação do nível de risco normalmente envolve a avaliação da probabilidade de ocorrência e do

impacto após a ocorrência de cada item de risco. Normalmente, a probabilidade de ocorrência significa a chance de que exista um possível problema no sistema testado.

O *Technical Test Analyst* contribui com a detecção e a compreensão do possível risco técnico para cada item de risco, enquanto o *Test Analyst* contribui com a compreensão do possível impacto comercial do problema se ele vir a ocorrer.

Entre os fatores genéricos que normalmente precisam ser levados em consideração estão:

- A complexidade da tecnologia;
- A complexidade da estrutura do código;
- O conflito entre *stakeholders* referente a requisitos técnicos;
- Problemas de comunicação decorrentes da distribuição geográfica da organização de desenvolvimento;
- Ferramentas e tecnologias;
- Pressões de tempo, recursos e gestão;
- Falta de atividades anteriores de garantia de qualidade;
- Grande número de alterações de requisitos técnicos;
- Detecção de grande número de defeitos referentes a características técnicas de qualidade;
- Problemas técnicos de interface e integração.

Em vista das informações disponíveis sobre o risco, o *Technical Test Analyst* define os níveis de risco de acordo com as diretrizes estabelecidas pelo *Test Manager*. Por exemplo, o *Test Manager* pode determinar que é preciso atribuir ao risco uma nota de 1 a 10, sendo que 1 representa o risco mais alto.

1.4 Mitigação de riscos

Durante o projeto, o *Technical Test Analyst* influencia como o teste reage aos riscos identificados. Isto geralmente envolve o seguinte:

- A redução do risco através da execução de testes mais importantes e da implementação de atividades adequadas de mitigação e contingências, como consta na estratégia e no plano de teste;
- A avaliação de riscos com base nas outras informações coletadas no decorrer do projeto e a utilização de tais informações para a implementação de ações de mitigação voltadas para a diminuição da probabilidade ou do impacto dos riscos previamente detectados e analisados.

2 Testes baseados em estruturas – 225 minutos

Palavras-chave

condição atômica, teste de condição, teste de fluxo de controle, teste de condição de decisão, teste de condição múltipla, teste de caminho, curto-circuito, teste de comando, técnica baseada em estruturas

Objetivos de aprendizagem dos testes baseados em estruturas

2.2 Teste de condição

TTA-2.2.1 (K2) Entender como realizar a cobertura de condições e por que ela pode ser um teste menos rigoroso do que a cobertura de decisões.

2.3 Teste de condição de decisão

TTA-2.3.1 (K3) Elaborar casos de teste mediante a aplicação da técnica de modelagem de testes referente ao teste de condição de decisão para definir um nível de cobertura.

2.4 Teste de cobertura de decisão de condição modificada

TTA-2.4.1 (K3) Elaborar casos de teste mediante a aplicação da técnica de modelagem de testes referente ao teste de cobertura de decisão de condição modificada para definir um nível de cobertura.

2.5 Teste de condição múltipla

TTA-2.5.1 (K3) Elaborar casos de teste mediante a aplicação da técnica de modelagem de testes referente ao teste de condição múltipla para definir um nível de cobertura.

2.6 Teste de caminho

TTA-2.6.1 (K3) Elaborar casos de teste mediante a aplicação da técnica de modelagem de testes referente ao teste de caminho.

2.7 Teste API

TTA-2.7.1 (K2) Entender a aplicabilidade do teste API e os tipos de defeitos detectados.

2.8 Seleção de técnica baseada em estrutura

TTA-2.8.1 (K4) Selecionar uma técnica baseada em estrutura adequada de acordo com determinada situação do projeto.

2.1 Introdução

Este capítulo descreve principalmente as técnicas de modelagem de testes baseados em estruturas, que também são conhecidas como técnicas de teste caixa-branca ou baseado em códigos. As técnicas utilizam o código, os dados, a arquitetura e / ou o fluxo do sistema como base para a modelagem de testes. Cada técnica específica permite que os casos de teste sejam extraídos sistematicamente e se concentrem em um aspecto específico da estrutura que será levada em consideração. As técnicas fornecem critérios de cobertura que terão que ser medidos e relacionados a um objetivo definido por cada projeto ou organização. A cobertura total não significa que todos os testes foram concluídos e, sim, que a técnica utilizada deixou de recomendar testes úteis para a estrutura considerada.

Com a exceção da cobertura de condição, as técnicas de modelagem de testes baseados em estruturas discutidas neste *syllabus* são mais rigorosas do que as técnicas de cobertura de comandos e decisões do *syllabus* do nível fundamental [ISTQB_FL_SYL].

As seguintes técnicas são consideradas neste *syllabus*:

- Teste de condição;
- Teste de condição de decisão;
- Teste de cobertura de decisão de condição modificada;
- Teste de condição múltipla;
- Teste de caminho;
- Teste API.

As primeiras quatro técnicas listadas acima se baseiam em predicados de decisões e, no geral, encontram o mesmo tipo de defeitos. Independentemente da complexidade de um predicado de decisão, será considerado VERDADEIRO ou FALSO, sendo que um caminho passará pelo código e o outro não. Um defeito é detectado quando o caminho previsto não é tomado porque o predicado de uma decisão complexa não realizou a avaliação conforme o esperado.

Em geral, as primeiras quatro técnicas são sucessivamente mais meticulosas. Exigem a definição de mais testes para conseguir a cobertura pretendida e encontrar instâncias mais sutis deste tipo de defeito.

Vide [Bath08], [Beizer90], [Beizer95], [Copeland03] e [Koomen06].

2.2 Teste de condição

Em comparação com o teste de decisão (desvio), que leva em consideração a decisão como um todo e avalia os resultados VERDADEIROS e FALSOS em casos de teste diferentes, os testes de condição levam em conta o processo decisório. Cada predicado de decisão é formado por uma ou mais condições atômicas simples e cada uma delas realiza avaliações com um valor booleano discreto. São logicamente combinadas para determinar o resultado da decisão. Os casos de teste devem avaliar cada condição atômica de ambas as formas para atingir este nível de cobertura.

Aplicabilidade

O teste de condição provavelmente só é interessante em termos abstratos por conta das dificuldades abaixo. No entanto, compreendê-lo é necessário para atingir níveis superiores de cobertura que se baseiam nele.

Limitações / dificuldades

Quando houver duas ou mais condições atômicas em uma decisão, uma escolha ruim na hora de selecionar os dados do teste durante a modelagem de testes pode gerar a cobertura de condição sem chegar a uma cobertura de decisão. Por exemplo, suponha-se o predicado de decisão “A e B”.

	A	**B**	**A e B**
Teste 1	FALSO	VERDADEIRO	FALSO
Teste 2	VERDADEIRO	FALSO	FALSO

Para atingir uma cobertura de condição de 100%, os dois testes exibidos na tabela abaixo devem ser executados. Embora os dois testes consigam uma cobertura de condição de 100%, não atingem a cobertura de decisão, já que, em ambos os casos, o predicado indica FALSO.

Quando uma decisão consiste em uma única condição atômica, o teste de condição é idêntico ao teste de decisão.

2.3 Teste de condição de decisão

O teste de condição de decisão especifica que o teste deve atingir uma cobertura de condição (*vide* acima) e exige que a cobertura de decisão (*vide* o *syllabus* do nível fundamental [ISTQB_FL_SYL]) também seja alcançada. Uma escolha ponderada dos valores dos dados de teste das condições atômicas pode fazer que este nível de cobertura seja atingido sem acrescentar outros casos de teste além dos necessários para alcançar a cobertura de condição.

O exemplo abaixo testa o mesmo predicado de decisão supracitado: “A e B”. A cobertura de condição de decisão pode ser atingida com o mesmo número de testes através da seleção de valores de teste diferentes.

	A	**B**	**A e B**
Teste 1	VERDADEIRO	VERDADEIRO	VERDADEIRO
Teste 2	FALSO	FALSO	FALSO

Portanto, esta técnica pode levar vantagem em termos de eficiência.

Aplicabilidade

Este nível de cobertura deve ser levado em consideração quando o código testado for importante, mas não crucial.

Limitações / dificuldades

Como talvez exija mais casos de teste do que testes no nível de decisão, pode ser problemático quando o prazo for apertado.

2.4 Teste de cobertura de decisão de condição modificada

Esta técnica propicia um nível mais alto de cobertura do fluxo de controle. Supondo-se um número N de condições atômicas únicas, a cobertura de decisão de condição modificada pode ser atingida normalmente em N+1 casos de teste únicos. A cobertura de decisão de condição modificada realiza a cobertura de condição de decisão, mas, depois, exige o cumprimento das seguintes condições:

1. Deve haver pelo menos um teste em que o resultado da decisão mudaria se a condição atômica X fosse VERDADEIRA;
2. Deve haver pelo menos um teste em que o resultado da decisão mudaria se a condição atômica X fosse FALSA;
3. Cada condição atômica diferente possui testes que atendem aos requisitos 1 e 2.

	A	B	C	(A ou B) e C
Teste 1	VERDADEIRO	FALSO	VERDADEIRO	VERDADEIRO
Teste 2	FALSO	VERDADEIRO	VERDADEIRO	VERDADEIRO
Teste 3	FALSO	FALSO	VERDADEIRO	FALSO
Teste 4	VERDADEIRO	FALSO	FALSO	FALSO

No exemplo acima, a cobertura de decisão é atingida (o resultado do predicado da decisão é tanto VERDADEIRO quanto FALSO) e a cobertura de condição é obtida (A, B e C são avaliados VERDADEIROS e FALSOS).

No teste 1, A é VERDADEIRO e a saída é VERDADEIRO. Se A mudasse para FALSO (como no teste 3, sendo que os outros valores permaneceriam inalterados), o resultado passaria a ser FALSO.

No teste 2, B é VERDADEIRO e a saída é VERDADEIRO. Se B mudasse para FALSO (como no teste 3, sendo que os outros valores permaneceriam inalterados), o resultado passaria a ser FALSO.

No teste 1, C é VERDADEIRO e a saída é VERDADEIRO. Se C mudasse para FALSO (como no teste 4, sendo que os outros valores permaneceriam inalterados), o resultado passaria a ser FALSO.

Aplicabilidade

Esta técnica é muito utilizada no setor de *software* aeroespacial e em muitos outros sistemas de segurança crítica. Deve ser utilizada na hora de lidar com *software* de segurança crítica em que qualquer falha pode provocar uma catástrofe.

Limitações / dificuldades

A obtenção da cobertura de decisão de condição modificada pode ser complicada quando houver várias ocorrências de um termo específico em uma expressão. Quando isto acontecer, diz-se que o termo está “acoplado”. Dependendo do comando da decisão no código, às vezes não é possível mudar o valor do termo acoplado de maneira que ele mesmo altera o resultado da decisão. Na hora de lidar com este problema, uma das abordagens disponíveis consiste em definir que apenas condições atômicas não acopladas podem ser

testadas no nível de cobertura de decisão de condição modificada. A outra abordagem consiste em analisar cada decisão em que o acoplamento ocorre caso a caso.

Alguns interpretadores e / ou linguagens de programação são concebidos para que sofram um curto-circuito na hora de avaliar um comando de decisão complexo no codigo. Isto é, o código de execução pode não avaliar a expressão inteira se o resultado da avaliação puder ser determinado somente após a avaliação de uma parte da expressão. Por exemplo, se a decisão "A e B" for avaliada, não há por quê avaliar B se A for FALSO. Nenhum valor de B pode mudar o valor final. Assim, o código pode economizar o tempo de execução ao não avaliar B. O curto-circuito pode afetar a capacidade de atingir a cobertura de decisão de condição modificada, já que alguns testes necessários podem não ser viáveis.

2.5 *Teste de condição múltipla*

Em casos raros, talvez seja necessário testar todas as combinações possíveis de valores que uma decisão pode conter. Este nível exaustivo de teste se chama cobertura de condição múltipla. O número de testes necessários depende do número de condições atômicas no comando da decisão e pode ser determinado através do cálculo de 2n, em que n é o número de condições atômicas não acopladas. Com o mesmo exemplo anterior, os seguintes testes são necessários para atingir a cobertura de condição múltipla:

	A	**B**	**C**	**(A ou B) e C**
Teste 1	VERDADEIRO	VERDADEIRO	VERDADEIRO	VERDADEIRO
Teste 2	VERDADEIRO	VERDADEIRO	FALSO	FALSO
Teste 3	VERDADEIRO	FALSO	VERDADEIRO	VERDADEIRO
Teste 4	VERDADEIRO	FALSO	FALSO	FALSO
Teste 5	FALSO	VERDADEIRO	VERDADEIRO	VERDADEIRO
Teste 6	FALSO	VERDADEIRO	FALSO	FALSO
Teste 7	FALSO	FALSO	VERDADEIRO	FALSO
Teste 8	FALSO	FALSO	FALSO	FALSO

Se a linguagem usar curto-circuito, o número de casos de teste reais frequentemente será reduzido, dependendo da ordem e do agrupamento de operações lógicas realizadas nas condições atômicas.

Aplicabilidade

Tradicionalmente, está técnica era utilizada para testar *software* incorporado que deveria rodar de maneira confiável sem panes durante longos períodos de tempo (por exemplo, esperava-se que os comutadores telefônicos durassem 30 anos). Este tipo de teste provavelmente substituirá o teste de cobertura de decisão de condição modificada em aplicativos muito importantes.

Limitações / dificuldades

Como o número de casos de teste pode ser extraído diretamente de uma tabela da verdade com todas as condições atômicas, este nível de cobertura pode ser facilmente determinado. No entanto, o número necessário de casos de teste permite que a cobertura de decisão de condição modificada seja aplicada na maioria das situações.

2.6 Teste de caminho

O teste de caminho consiste em identificar os caminhos no código e, então, criar testes para cobri-los. Conceitualmente, seria útil testar todos os caminhos únicos no sistema. Em um sistema não trivial, contudo, o número de casos de teste poderia ser excessivamente grande devido à natureza das estruturas em *loop*.

Deixando de lado a questão do *looping* indefinido, é possível realizar alguns testes de caminho. Para aplicar esta técnica, [Beizer90] recomenda a criação de testes que seguem muitos caminhos através do módulo da entrada à saída. Para simplificar o que poderia ser uma tarefa complexa, ele recomenda que isto seja feito sistematicamente com o seguinte procedimento:

1. Como primeiro caminho, escolher o caminho mais simples e funcionalmente sensível da entrada à saída;
2. Escolher cada caminho seguinte como pequena variação do caminho anterior. Tentar mudar apenas um desvio do caminho, que é diferente em cada teste sucessivo. Favorecer caminhos curtos em vez de caminhos longos, quando possível. Favorecer caminhos que fazem sentido funcional em detrimento dos que não fazem sentido funcional;
3. Escolher caminhos que não fazem sentido funcional apenas quando se trata de uma exigência da cobertura. Nesta regra, Beizer observa que tais caminhos podem ser externos e devem ser questionados;
4. Usar a intuição na hora de escolher caminhos (isto é, quais caminhos têm mais chances de ser executados).

Repare que alguns segmentos de caminho provavelmente serão executados mais de uma vez com esta estratégia. A questão principal desta estratégia é a de testar todo possível desvio no código pelo menos uma vez e talvez muitas vezes.

Aplicabilidade

A realização de testes em caminhos parciais – como já se definiu acima – acontece frequentemente com *software* de segurança crítica. É um dos outros bons métodos discutidos neste capítulo porque examina os caminhos no *software* e não só o processo decisório.

Limitações / dificuldades

Embora seja possível utilizar um gráfico de fluxo de controle para determinar os caminhos, realistamente, há necessidade de uma ferramenta para calculá-los em módulos complexos.

Cobertura

A criação de testes suficientes para cobrir todos os caminhos (desconsiderando os *loops*) garante a realização da cobertura de comandos e desvios. O teste de caminho realiza testes mais meticulosos do que a cobertura de desvios, com um aumento relativamente pequeno do número de testes. [NIST 96]

2.7 Teste API

Uma interface de programação de aplicativos (API, na sigla em inglês) é um código que possibilita a comunicação entre processos, programas e / ou sistemas diferentes. As APIs são frequentemente utilizadas em relacionamentos entre clientes e servidores em que um processo propicia algum tipo de funcionalidade a outros processos.

De certa maneira, o teste API se parece muito com o teste com a interface gráfica de usuário (GUI, na sigla em inglês). A avaliação de valores de entrada e dados de retorno é enfatizada.

O teste negativo é frequentemente crucial na hora de lidar com APIs. Os programadores que usam APIs para acessar serviços fora do código podem tentar utilizar interfaces API de maneiras que não foram previstas. Isto significa que o tratamento robusto de erros é essencial para evitar operações incorretas. Os testes combinatórios com muitas interfaces diferentes podem ser necessários porque as APIs são frequentemente utilizadas com outras APIs e porque uma única interface pode conter diversos parâmetros cujos valores podem ser combinados de muitas maneiras.

As APIs frequentemente não são acopladas rigorosamente, o que resulta na possibilidade muito real de transações perdidas ou *glitches* no *timing*. Isto requer testes meticulosos nos mecanismos de recuperação e novas tentativas. Uma organização que fornece uma interface API deve garantir que todos os serviços possuam uma disponibilidade muito alta. Frequentemente isto exige testes de confiabilidade rigorosos do editor da API e suporte à infraestrutura.

Aplicabilidade

Os testes API ficam mais importantes à medida que os sistemas são distribuídos ou utilizam processamento remoto como forma de descarregar trabalhos em outros processadores. Entre os exemplos estão chamadas de sistemas operacionais, arquiteturas orientadas a serviços (SOA, na sigla em inglês), chamadas remotas de procedimento (RPC, na sigla em inglês), serviços de rede e, praticamente, todos os aplicativos distribuídos. O teste API diz respeito especificamente aos testes com sistemas de sistemas.

Limitações / dificuldades

Testar uma API diretamente costuma exigir que o *Technical Test Analyst* utilize ferramentas especializadas. Como normalmente não há nenhuma interface gráfica direta ligada a uma API, as ferramentas podem ser necessárias para configurar o ambiente inicial, reunir os dados, invocar a API e determinar o resultado.

Cobertura

O teste API é uma descrição de um tipo de teste e não denota nenhum nível específico de cobertura. No mínimo, o teste API deve incluir exercícios com todas as chamadas para a API e todos os valores válidos e razoáveis e inválidos.

Tipos de defeitos

Os tipos de defeitos que podem ser detectados pelos testes API são bem diferentes. Problemas na interface são comuns, assim como problemas no tratamento de dados, problemas de *timing*, perda de transações e transações em duplicidade.

2.8 Seleção de técnica baseada em estrutura

O contexto do sistema testado determinará o nível de cobertura de teste baseado em estrutura que deve ser alcançado. Quanto mais importante for o sistema, maior será o nível de cobertura necessário. Em geral, quanto maior o nível de cobertura necessário, mais tempo e recursos serão necessários para atingir tal nível.

Às vezes, o nível de cobertura exigido pode ser extraído de normas pertinentes que se aplicam ao sistema de *software*. Por exemplo, se o *software* for utilizado em um ambiente aéreo, talvez seja necessária a adequação à norma DO-178B (na Europa, ED-12B). Esta norma contém estas cinco condições de falha:

A. **Catastróficas**: as falhas podem provocar a ausência de funções críticas necessárias para pilotar ou aterrissar a aeronave com segurança;
B. **Perigosas**: as falhas podem causar um impacto negativo na segurança ou no desempenho;
C. **Importantes**: as falhas são consideráveis, mas menos graves que A ou B;
D. **Menores**: as falhas são perceptíveis, mas com um impacto menor do que C;
E. **Sem efeito**: as falhas não afetam a segurança.

Se o sistema de *software* cair no nível A, a cobertura de decisão de condição modificada deve ser testada. Se for o nível B, a cobertura de nível de decisão deve ser testada, embora a cobertura de decisão de condição modificada seja opcional. O nível C exige, pelo menos, a cobertura de comando.

Igualmente, a IEC-61508 é uma norma internacional que regula a segurança funcional de sistemas programáveis e eletrônicos relacionados à segurança. Esta norma foi adaptada a muitas áreas diferentes, inclusive o setor automotivo, o setor ferroviário, os processos fabris, as usinas nucleares e o maquinário. A criticidade é definida com um contínuo graduado do nível de integridade de segurança (SIL, na sigla em inglês), sendo que 1 é o menos crítico e 4 o mais crítico. A cobertura é recomendada da seguinte maneira:

1. Cobertura recomendada de comandos e desvios;
2. Cobertura de comandos altamente recomendada e cobertura de desvios recomendada;
3. Cobertura de comandos e desvios altamente recomendada;
4. Cobertura de decisão de condição modificada altamente recomendada.

Em sistemas modernos, é raro que o processamento seja todo realizado em um único sistema. O teste API deve ser instituído quando parte do processamento for realizado remotamente. A criticidade do sistema deve determinar o esforço investido nos testes API.

Como sempre, o contexto do sistema de *software* testado deve orientar o *Technical Test Analyst* nos métodos utilizados nos testes.

3 Técnicas analíticas – 255 minutos

Palavras-chave

análise de fluxo de controle, complexidade ciclomática, análise de fluxo de dados, par definição-utilização, análise dinâmica, vazamento de memória, teste de integração por pares, teste de integração de vizinhança, análise estática, ponteiro perdido

Objetivos de aprendizagem das técnicas analíticas

3.2 Análise estática

TTA-3.2.1 (K3) Utilizar a análise de fluxo de controle para detectar se o código possui alguma anomalia no fluxo de controle;

TTA-3.2.2 (K3) Utilizar a análise de fluxo de dados para detectar se o código possui alguma anomalia no fluxo de dados;

TTA-3.2.3 (K3) Propor melhorias na manutenibilidade do código através da aplicação de análises estáticas;

TTA-3.2.4 (K2) Explicar o uso de gráficos de chamadas para estabelecer estratégias de teste de integração.

3.3 Análise dinâmica

TTA-3.3.1 (K3) Especificar metas que devem ser atingidas através da utilização da análise dinâmica.

3.1 Introdução

Existem dois tipos de análises: a **análise estática** e a **análise dinâmica**.

A **análise estática** (item 3.2) abrange o teste analítico que ocorre sem a execução de *software*. Como o *software* não está sendo executado, é examinado ou por uma ferramenta ou por uma pessoa para determinar se será processado corretamente quando for executado. Este ponto de vista estático do *software* permite uma análise detalhada sem ter que criar os dados e as pré-condições que provocariam a ocorrência do cenário.

As diferentes formas de revisão atinentes ao *Technical Test Analyst* são discutidas no capítulo 5.

A **análise dinâmica** (item 3.3) exige a execução real do código e serve para detectar falhas de codificação que são detectadas com maior facilidade quando o código estiver sendo executado (por exemplo, vazamentos de memória). A análise dinâmica, assim como a análise estática, pode depender de ferramentas ou de um indivíduo para monitorar a execução do sistema em busca de indicadores como crescimento rápido da memória.

3.2 Análise estática

O objetivo da análise estática é o de detectar falhas reais ou possíveis na arquitetura do código e do sistema e aprimorar sua manutenibilidade. Geralmente, a análise estática é suportada por ferramentas.

3.2.1 Análise de fluxo de controle

A análise de fluxo de controle é a técnica estática em que o fluxo de controle através de um programa é analisado, quer com um gráfico, quer com um ferramenta de fluxo de controle. Muitas anomalias podem ser detectadas em um sistema com esta técnica, inclusive *loops* mal concebidos (por exemplo, com vários pontos de entrada), alvos ambíguos de chamadas de função em certas linguagens (por exemplo, Scheme), o sequenciamento incorreto de operações *etc*.

Um dos usos mais comuns da análise de fluxo de controle é a determinação da complexidade ciclomática. O valor da complexidade ciclomática é um número inteiro positivo que representa o número de caminhos independentes em um gráfico com muitas conexões, com *loops* e iterações ignorados assim que tiverem sido percorridos uma vez. Cada caminho independente, da entrada à saída, representa um caminho único através do módulo. Cada caminho único deve ser testado.

Geralmente, o valor da complexidade ciclomática consiste em entender a complexidade geral de um módulo. A teoria de Thomas McCabe [McCabe 76] era a de que quanto mais complexo for o sistema, mais difícil seria mantê-lo e mais defeitos ele conteria. Ao longo dos anos, muitos estudos observaram esta correlação entre complexidade e o número de defeitos contidos. O Instituto Nacional de Metrologia, Normalização e Qualidade Industrial dos EUA (NIST, na sigla em inglês) recomenda um valor máximo de complexidade de 10. Qualquer módulo que tiver uma complexidade maior do que essa poderá ser dividido em vários módulos.

3.2.2 Análise de fluxo de dados

A análise de fluxo de dados cobre várias técnicas que coletam informações sobre o uso de variáveis em um sistema. O ciclo de vida das variáveis (isto é, onde são declaradas, definidas, lidas, avaliadas e destruídas) é examinado, já que as anomalias podem ocorrer durante qualquer uma destas operações.

Uma das técnicas comuns é a chamada notação definição-utilização, segundo a qual o ciclo de vida de cada variável é dividido em três ações atômicas diferentes:

- **d**: quando a variável for declarada, definida ou inicializada;
- **u**: quando a variável for utilizada ou lida em um cálculo ou um predicado de decisão;
- **k**: quando a variável for eliminada ou destruída ou sair do escopo.

Estas três ações atômicas são combinadas para formarem pares (pares definição-utilização) para ilustrarem o fluxo de dados. Por exemplo, um caminho que representa um fragmento do código em que uma variável de dados é definida e, então, utilizada em seguida.

Entre as possíveis anomalias nos dados estão a realização de uma ação correta em uma variável no momento errado ou a realização de uma ação incorreta nos dados de uma variável. Entre as anomalias estão:

- Atribuir um valor inválido a uma variável;
- Não conseguir atribuir um valor a uma variável antes de utilizá-la;
- Seguir um caminho incorreto devido a um valor incorreto no predicado de controle;
- Tentar utilizar uma variável após sua destruição;
- Elencar uma variável quando estiver fora de escopo;
- Declarar e destruir uma variável sem utilizá-la;
- Redefinir uma variável antes de ter sido utilizada;
- Não eliminar uma variável dinamicamente alocada (o que provoca um possível vazamento de memória);
- Modificar uma variável, o que leva a efeitos colaterais inesperados (por exemplo, repercussões ao alterar a variável global sem levar em conta todos os usos da variável).

A linguagem de desenvolvimento que é utilizada pode orientar as regras usadas na análise de fluxo de dados. As linguagens de programação permitem que o programador realize certas ações com variáveis que não são ilegais, mas que podem provocar no sistema um comportamento diferente daquele esperado pelo programador em certas circunstâncias. Por exemplo, uma variável pode ser definido duas vezes sem ser utilizada quando certo caminho é seguido. A análise do fluxo de dados frequentemente chamará tais usos suspeitos. Embora possa ser um uso legal dos recursos de atribuição de variáveis, pode levar a futuros problemas de manutenibilidade no código.

O teste de fluxo de dados "usa o gráfico do fluxo de controle para explorar o que pode acontecer com os dados" [Beizer90] e, portanto, detecta defeitos diferentes dos encontrados pelos testes de fluxo de controle. O *Technical Test Analyst* deve incluir esta técnica na hora de planejar os testes, já que muitos destes defeitos provocam falhas intermitentes que são difíceis de detectar ao realizar testes dinâmicos.

No entanto, a análise de fluxo de dados é uma técnica estática. Ela pode ignorar alguns problemas que ocorrem nos dados durante o tempo de execução do sistema. Por exemplo, a variável de dados estáticos pode conter um ponteiro no arranjo dinamicamente criado que nem sequer existe até o tempo de execução. A utilização de vários processadores e a versatilidade preventiva podem criar condições de corrida que não serão detectadas pela análise de fluxo de dados ou fluxo de controle.

3.2.3 Utilização de análise estática para o aprimoramento da manutenibilidade

A análise estática pode ser aplicada de várias maneiras com a finalidade de aprimorar a manutenibilidade do código, a arquitetura e os *sites*.

Um código mal escrito, não comentado e não estruturado tende a ser mais difícil de manter. Talvez os desenvolvedores precisem trabalhar mais para localizar e analisar os defeitos no código e a modificação do código para a correção de um defeito ou o acréscimo de uma nova característica pode resultar na introdução de outros defeitos.

A análise estática é utilizada com o suporte de ferramentas para aprimorar a manutenibilidade do código através da verificação da complacência com normas e diretrizes de codificação. As normas e as diretrizes descrevem as práticas necessárias de codificação, como nomenclatura, comentários, indentação e modulação de códigos. No geral, as ferramentas de análise estática apontam advertências e não erros, muito embora o código possa estar sintaticamente correto.

Os *designs* modulares geralmente resultam em um código mais fácil de manter. As ferramentas de análise estática suportam o desenvolvimento de códigos modulares das seguintes maneiras:

- Buscam o código repetido. Estas partes do código podem ser candidatas à refatoração em módulos (embora os custos do tempo de execução impostos pelas chamadas do módulo possam ser um problema para sistemas em tempo real);
- Geram métricas que são indicadores valiosos da modulação do código. Entre elas estão medidas de acoplamento e coesão. Um sistema com boa manutenibilidade tem mais chances de possuir um baixo nível de acoplamento (o que varia de acordo com o grau de interdependência dos módulos durante a execução) e um alto nível de coesão (o que varia de acordo com a autonomia e o foco dos módulos em uma única tarefa);
- Em códigos orientados a objetos, indicam onde objetos derivados possuem visibilidade demais ou de menos em classes-pai;
- No código ou na arquitetura, salientam as áreas com alto nível de complexidade estrutural, o que geralmente é considerado um indício de manutenibilidade ruim e um potencial maior de contenção de falhas. Os níveis aceitáveis de complexidade ciclomática (*vide* o item 3.2.1) podem ser especificados em diretrizes para garantir que o código seja desenvolvido de maneira modular com a manutenibilidade e a prevenção de defeitos em mente. Um código com altos níveis de complexidade ciclomática pode virar candidato à modulação.

A manutenção de um *site* também pode ser suportada por ferramentas de análise estática. É preciso verificar se a estrutura arborescente do *site* é bem equilibrada ou se há algum desequilíbrio que levará a:

- Uma maior dificuldade nas tarefas de teste;
- Um aumento da carga de trabalho de manutenção;
- Uma maior dificuldade para a navegação do usuário.

3.2.4 Gráficos de chamadas

Os gráficos de chamadas são uma representação estática da complexidade da comunicação. São gráficos direcionados em que os nós representam unidades de programa e as bordas representam a comunicação entre as unidades.

Os gráficos de chamadas podem ser utilizados nos testes de unidade em que funções ou métodos diferentes fazem chamadas entre si, em testes de integração e de sistema em que módulos diferentes chamam um ao outro ou em testes de integração de sistemas em que sistemas diferentes chamam um ao outro.

Os gráficos de chamadas podem ser utilizadas para as seguintes finalidades:

- Modelar testes que chamem um módulo ou um sistema específico;

- Definir o número de locais no *software* de que um módulo ou um sistema é chamado;
- Avaliar a estrutura do código e da arquitetura do sistema;
- Sugerir a ordem da integração (integração por pares ou de vizinhança), que é discutida mais detalhadamente abaixo.

No *syllabus* do nível fundamental [ISTQB_FL_SYL], duas categorias diferentes de testes de integração foram discutidas: uma categoria incremental (*top-down*, *bottom-up etc.*) e não incremental (*big bang*). Diz-se que os métodos incrementais são preferidos porque introduzem o código pouco a pouco, facilitando, assim, o isolamento das falhas, já que a quantidade de código envolvida é limitada.

Neste *syllabus* do nível avançado, três outros métodos não incrementais que usam gráficos de chamadas são introduzidos. Podem ser preferíveis em relação a métodos incrementais, que provavelmente exigirão outros pacotes para a conclusão do teste e a formulação de um código não entregável para prestar suporte aos testes. Os três métodos são:

- O teste de integração por pares (que não deve ser confundido com o teste alternado, uma técnica de teste caixa-preta) visa a pares de componentes que funcionam juntos de acordo com o gráfico de chamadas do teste de integração. Embora este método reduza só um pouco o número de pacotes, diminui a quantidade necessária de código preparado para teste;
- A integração de vizinha testa todos os nós relacionados a determinado nó como base para os testes de integração. Todos os nós predecessores e sucessores de um nó específico no gráfico de chamadas servem de base para o teste;
- A abordagem do predicado de modelagem de McCabe utiliza a teoria da complexidade ciclomática aplicada nos gráficos de chamadas referentes a módulos. Isto exige a elaboração de um gráfico de chamadas que mostre as diferentes maneiras pelas quais os módulos conseguem chamar um ao outro, inclusive:
  - Chamadas incondicionais: a chamada de um módulo para outro sempre acontece;
  - Chamadas condicionais: a chamada de um módulo para outro acontece às vezes;
  - Chamadas condicionais mutuamente exclusivas: um módulo chama outro (e apenas aquele) de uma série de módulos diferentes;
  - Chamadas iterativas: um módulo chama outro pelo menos uma vez, mas pode chamá-lo várias vezes;
  - Chamadas iterativas condicionais: um módulo pode chamar outro de zero a muitas vezes.

Após a criação do gráfico de chamadas, a complexidade da integração é calculada e os testes são criados para cobrir o gráfico.

Consulte [Jorgensen07] para obter mais informações sobre a utilização dos gráficos de chamadas e dos testes de integração por pares.

3.3 Análise dinâmica

3.3.1 Panorama

A análise dinâmica serve para detectar falhas quando os sintomas não forem visíveis imediatamente. Por exemplo, vazamentos de memória podem ser detectados através da análise estática (ao encontrar o código que aloca, mas não libera memória), mas, com a análise dinâmica, aparecem na hora.

As falhas que não forem imediatamente reprodutíveis podem afetar muito os trabalhos de teste e a

capacidade de lançar ou utilizar o *software* produtivamente. Tais falhas podem ser provocadas por vazamentos de memória, pelo uso incorreto de ponteiros e por outras corrupções (por exemplo, da pilha do sistema) [Kaner02]. Devido à natureza das falhas, que podem incluir uma piora gradual do desempenho do sistema ou até mesmo panes no sistema, as estratégias de teste devem levar em conta os riscos ligados a tais defeitos e, se for o caso, realizar análises dinâmicas para reduzi-los (normalmente com ferramentas). Como estas falhas frequentemente são as mais caras de detectar e corrigir, recomenda-se a realização de análises dinâmicas no início do projeto.

A análise dinâmica pode ser aplicada para conseguir o seguinte:

- Impedir a ocorrência de falhas através da detecção de ponteiros perdidos e perda de memória do sistema;
- Analisar falhas no sistema que não podem ser facilmente reproduzidas;
- Avaliar o comportamento da rede;
- Aprimorar o desempenho do sistema ao fornecer informações sobre o comportamento do sistema durante o tempo de execução.

As análises dinâmicas também podem ser realizadas em qualquer nível de teste e exigem habilidades técnicas e de sistemas para:

- Especificar os objetivos de teste da análise dinâmica;
- Determinar o momento certo para iniciar e interromper a análise;
- Analisar os resultados.

Durante os testes do sistema, as ferramentas de análise dinâmica podem ser utilizadas mesmo se o *Technical Test Analyst* tiver habilidades técnicas mínimas. As ferramentas utilizadas normalmente criam registros abrangentes que podem ser analisados por pessoas com as habilidades técnicas necessárias.

3.3.2 Detecção de vazamentos de memória

Vazamentos de memória ocorrem quando as áreas da memória (RAM) disponível de um programa são alocadas pelo programa, mas não são liberadas depois quando deixarem de ser necessárias. Esta área da memória fica alocada e não fica disponível para reutilização. Quando isto ocorrer com frequência ou em situações em que há pouca memória, o programa pode ficar sem memória utilizável. Historicamente, a manipulação de memória cabe ao programador. Quaisquer áreas da memória dinamicamente alocadas tinham que ser liberadas pelo programa que realizou a alocação de acordo com o escopo correto para evitar um vazamento de memória. Muitos ambientes modernos de programação incluem a "coleta do lixo" automática ou semiautomática, na qual a memória alocada é liberada sem a intervenção direta do programador. O isolamento de vazamentos de memória pode ser muito difícil quando a memória alocada for liberada pela coleta de lixo automática.

Os vazamentos de memória provocam problemas que se desenvolvem com o tempo e nem sempre são imediatamente óbvios. Isto pode ocorrer se, por exemplo, o *software* houver sido instalado recentemente ou o sistema houver sido reinicializado, o que acontece frequentemente durante os testes. Por estes motivos, frequentemente os efeitos negativos dos vazamentos de memória podem ser percebidos inicialmente quando o programa está em produção.

Os sintomas dos vazamentos de memória representam uma piora constante do tempo de resposta do sistema, o que pode, em última instância, provocar uma falha no sistema. Embora tais falhas possam ser resolvidas através da reinicialização do sistema, isto nem sempre é prático ou até mesmo possível.

Muitas ferramentas de análise dinâmica identificam áreas no código em que ocorrem vazamentos de memória para que possam ser corrigidos. Simples monitores de memória também podem ser utilizados para saber se a disponibilidade de memória cai com o tempo, embora uma análise ulterior ainda seja necessária para determinar as causas exatas da queda.

Existem outras fontes de vazamentos que também devem ser levadas em consideração. Entre os exemplos estão descritores de arquivos, semáforos e *pools* de conexões de recursos.

3.3.3 Detecção de ponteiros perdidos

Em um programa, um ponteiro perdido é aquele que não deve ser utilizado. Por exemplo, um ponteiro perdido pode ter "perdido" o objeto ou a função para o qual deveria estar apontando ou não aponta a área de memória prevista (por exemplo, aponta uma área que está além dos limites alocados de um arranjo). Quando um programa usa ponteiros perdidos, diversas consequências podem ocorrer:

- O programa pode funcionar como esperado. Isto acontece quando o ponteiro perdido acessa a memória que não está sendo utilizada pelo programa e está, na teoria, "livre" e / ou contém um valor razoável;
- O programa pode travar. Neste caso, o ponteiro perdido pode ter ocasionado a utilização incorreta de uma parte da memória, o que é sumamente importante para a execução do programa (por exemplo, o sistema operacional);
- O programa não funciona corretamente porque os objetos exigidos pelo programa não podem ser acessados. Nestas condições, o programa pode continuar funcionando, embora uma mensagem de erro possa aparecer;
- Os dados no local da memória podem ser corrompidos pelo ponteiro e valores incorretos podem ser utilizados em seguida.

Quaisquer alterações feitas no uso da memória do programa (por exemplo, um novo pacote após uma alteração no *software*) podem provocar qualquer uma das quatro consequências listadas acima. Isto é particularmente crucial quando, inicialmente, o programa tem um desempenho esperado apesar do uso de ponteiros perdidos e, então, trava de repente (talvez até mesmo na produção) após uma alteração no *software*. Vale lembrar que tais falhas são, frequentemente, sintomas de um defeito subjacente (isto é, o ponteiro perdido). (*Vide* a *Lição 74* em [Kaner02].) As ferramentas podem ajudar a identificar ponteiros perdidos quando são utilizadas pelo programa, independentemente do impacto sobre a execução do programa. Alguns sistemas operacionais possuem funções incorporadas para verificar infrações no acesso à memória durante o tempo de execução. Por exemplo, o sistema operacional pode disparar uma exceção quando um aplicativo tentar acessar um local da memória que fica fora da área de memória permitida do aplicativo.

3.3.4 Análise de desempenho

A análise dinâmica não é só útil para a detecção de falhas. Com a análise dinâmica do desempenho do programa, as ferramentas contribuem com a identificação de gargalos de desempenho e geram diversas métricas de desempenho, que podem ser utilizadas pelo desenvolver para ajustar o desempenho do sistema. Por exemplo, é possível fornecer informações sobre quantas vezes um módulo é chamado durante a execução. Os módulos que forem chamados frequentemente seriam candidatos ao aprimoramento de desempenho.

Ao reunir as informações sobre o comportamento dinâmico do *software* com as informações obtidas de

gráficos de chamadas durante a análise estática (*vide* o item 3.2.4), o testador também consegue identificar os módulos que poderiam ser candidatos a testes detalhados e abrangentes (por exemplo, módulos que são chamados frequentemente e possuem muitas interfaces).

A análise dinâmica do desempenho de um programa é realizada frequentemente ao mesmo tempo em que testes de sistema são realizados, embora também possa ser feita ao testar um único subsistema nas primeiras fases do teste com ambientes preparados para testes.

4 Características de qualidade dos testes técnicos – 405 minutos

Palavras-chave

adaptabilidade, analisabilidade, modificabilidade, coexistência, eficiência, instalabilidade, teste de manutenibilidade, maturidade, teste de aceitação operacional, perfil operacional, teste de desempenho, teste de portabilidade, teste de recuperabilidade, modelo de crescimento da confiabilidade, teste de confiabilidade, substitutibilidade, teste de utilização de recursos, robustez, teste de segurança, estabilidade, testabilidade

Objetivos de aprendizagem das características de qualidade de testes técnicos

4.2 Questões gerais de planejamento

TTA-4.2.1 (K4) Em relação a determinado projeto e sistema testado, analisar os requisitos não funcionais e redigir as respectivas seções do plano de teste.

4.3 Teste de segurança

TTA-4.3.1 (K3) Definir a abordagem e modelar os casos de teste de alto nível para os testes de segurança.

4.4 Teste de confiabilidade

TTA-4.4.1 (K3) Definir a abordagem e modelar os casos de teste de alto nível para a confiabilidade, uma característica de qualidade, e as subcaracterísticas correspondentes de acordo com a norma ISO 9126.

4.5 Teste de desempenho

TTA-4.5.1 (K3) Definir a abordagem e modelar os perfis operacionais de alto nível para os testes de desempenho.

Objetivos de aprendizagem comuns

Os seguintes objetivos de aprendizagem dizem respeito ao conteúdo discutido em mais de uma seção deste capítulo.

TTA-4.x.1 (K2) Compreender e explicar os motivos da inclusão de testes de manutenibilidade, de portabilidade e de utilização de recursos em uma estratégia e / ou abordagem de teste;

TTA-4.x.2 (K3) Considerando determinado risco de produto, definir o/s tipo/s de testes não funcionais mais adequados;

TTA-4.x.3 (K2) Compreender e explicar as etapas no ciclo de vida de um aplicativo em que os testes não funcionais devem ser aplicados;

TTA-4.x.4 (K3) Em determinado cenário, definir os tipos de defeitos esperados com a utilização de tipos de testes não funcionais.

4.1 Introdução

Em geral, o *Technical Test Analyst* se concentra em testar *como* o produto funciona e não os aspectos funcionais *daquilo* que faz. Os testes podem acontecer em qualquer nível de teste. Por exemplo, durante os testes de componente com sistemas em tempo real e incorporados, a realização de *benchmarking* de desempenho e a utilização de recursos de teste são importantes. Durante o teste de sistema e o teste de aceitação operacional, os testes de aspectos de confiabilidade, como a recuperabilidade, são adequados. Neste nível, os testes são voltados para um sistema específico, isto é, combinações de *hardware* e *software*. O sistema específico testado pode incluir diversos servidores, clientes, bancos de dados, redes e outros recursos. Independentemente do nível do teste, os testes devem ser realizados de acordo com as prioridades de risco e os recursos disponíveis.

A descrição das características de qualidade do produto estipuladas na ISO 9126 serve de guia para a descrição das características. Outras normas, como a série ISO 25000 (que suplantou a ISO 9126), também podem ser úteis. As características de qualidade da ISO 9126 são divididas em características e cada uma delas pode ter subcaracterísticas. Aparecem na tabela abaixo, juntamente com uma indicação de qual característica / subcaracterística é coberta pelos *syllabi* do *Test Analyst* e do *Technical Test Analyst*.

Característica	Subcaracterísticas	Test Analyst	Technical Test Analyst
Funcionalidade	Acurácia, adequação, interoperabilidade, complacência	X	
	Segurança		X
Confiabilidade	Maturidade (robustez), tolerância a falhas, recuperabilidade, complacência		X
Usabilidade	Entendibilidade, assimilabilidade, operabilidade, atratividade, complacência	X	
Eficiência	Desempenho (comportamento relacionado a tempo), utilização de recursos, complacência		X
Manutenibilidade	Analisabilidade, modificabilidade, estabilidade, testabilidade, complacência		X
Portabilidade	Adaptabilidade, instalabilidade, coexistência, substitutibilidade, complacência		X

Embora tal alocação de trabalho possa variar em organizações diferentes, é seguida nestes *syllabi* da ISTQB.

A subcaracterística da complacência aparece em cada uma das características de qualidade. Em relação a certos ambientes regulados ou de segurança crítica, cada característica de qualidade pode ter que cumprir certos regulamentos e normas específicos. Como as normas podem variar bastante dependendo do setor, não

serão discutidas aprofundadamente aqui. Se o *Technical Test Analyst* estiver trabalhando em um ambiente afetado pelos requisitos de complacência, é importante compreendê-los e garantir que tanto o teste quanto a documentação do teste atendam aos requisitos de complacência.

Em relação a todas as características e subcaracterísticas de qualidade discutidas nesta seção, os riscos comuns devem ser reconhecidos para que uma estratégia de teste adequada possa ser formada e documentada. O teste com características de qualidade exige foco específico no tempo do ciclo de vida, na disponibilidade necessária de ferramentas, *software* e documentos e na *expertise* técnica. Sem o planejamento de uma estratégia para lidar com cada característica e suas necessidades de teste únicas, o testador pode não ter tempo suficiente para o planejamento, a preparação e a execução de testes no cronograma. [Bath08] Alguns destes testes, por exemplo, o teste de desempenho, podem exigir muito planejamento, equipamento dedicado, ferramentas específicas, habilidades especializadas em testes e, na maioria dos casos, bastante tempo.

Os testes com características e subcaracterísticas de qualidade devem ser integrados ao cronograma geral de teste e recursos suficientes devem ser alocados aos trabalhos. Cada uma das áreas possui necessidades específicas, objetiva problemas específicos e pode acontecer em momentos diferentes durante o ciclo de vida do *software*, como se discute nas seções abaixo.

Enquanto o *Test Manager* se responsabiliza pela compilação e pela divulgação do resumo de métricas sobre as características e as subcaracterísticas de qualidade, o *Test Analyst* ou o *Technical Test Analyst* (de acordo com a tabela acima) coleta as informações referentes a cada métrica.

As medições das características de qualidade coletadas nos testes de pré-produção pelo *Technical Test Analyst* podem servir de base para acordos de nível de serviço (SLAs, na sigla em inglês) entre o fornecedor e os *stakeholders* (por exemplo, clientes, operadores) do sistema do *software*. Em alguns casos, os testes podem continuar sendo executados depois de o *software* entrar em produção, frequentemente por parte de uma equipe ou uma organização diferente. Normalmente, isto vale para os testes de eficiência e confiabilidade, que podem gerar resultados no ambiente de produção diferentes dos do ambiente de teste.

4.2 Questões gerais de planejamento

O não planejamento de testes não funcionais pode colocar em risco o sucesso de um aplicativo. O *Technical Test Analyst* pode receber do *Test Manager* uma solicitação de identificação dos principais riscos referentes às características de qualidade relevantes (*vide* tabela no item 4.1) e de tratamento de quaisquer problemas de tratamento ligados aos testes propostos. Podem ser utilizados na criação do plano-mestre de teste. Os seguintes fatores gerais são levados em consideração na hora de realizar estas tarefas:

- Os requisitos dos *stakeholders*;
- A aquisição das ferramentas necessárias e o treinamento;
- Os requisitos do ambiente de teste;
- As considerações organizacionais;
- As considerações de segurança de dados.

4.2.1 Requisitos dos stakeholders

Os requisitos não funcionais frequentemente são mal especificados ou nem sequer existem. Na etapa de planejamento, o *Technical Test Analyst* deve conseguir obter níveis de expectativa relacionados às características técnicas de qualidade dos *stakeholders* afetados e avaliar os riscos que representam.

Uma das abordagens comuns consiste em supor que, se o cliente estiver satisfeito com a versão existente do sistema, ele continuará satisfeito com as novas versões, contanto que os níveis de qualidade atingidos sejam mantidos. Isto permite que a versão existente do sistema sirva de *benchmark*. Esta abordagem pode ser particularmente útil para algumas características de qualidade não funcionais, como o desempenho, pois os *stakeholders* podem ter dificuldades na hora de especificarem seus requisitos.

É recomendável a obtenção de vários pontos de vista ao captar os requisitos não funcionais. Devem ser coletados junto aos *stakeholders*, como clientes, usuários, funcionários operacionais e o pessoal da manutenção, senão, alguns requisitos serão provavelmente desconsiderados.

4.2.2 Aquisição das ferramentas necessárias e treinamento

As ferramentas ou os simuladores comerciais são particularmente relevantes para testes de desempenho e certos testes de segurança. O *Technical Test Analyst* deve estimar os custos e os prazos envolvidos na aquisição, no aprendizado e na implementação das ferramentas. Quando as ferramentas especializadas forem utilizadas, o planejamento deve incluir o aprendizado da utilização das novas ferramentas e / ou o custo de contratar profissionais terceirizados especializados nas ferramentas.

O desenvolvimento de um simulador complexo pode representar um projeto de desenvolvimento em si e deve ser planejado desta forma.

Em particular, o teste e a documentação da ferramenta desenvolvida devem ser levados em consideração no cronograma e no plano de recursos. Devem existir verbas e tempo suficientes para o *upgrade* e o reteste do simulador à medida que o produto da simulação mudar. O planejamento para os simuladores que serão utilizados em aplicativos de segurança crítica deve levar em conta o teste de aceitação e a possível certificação do simulador por uma entidade independente.

4.2.3 Requisitos do ambiente de teste

Muitos testes técnicos (por exemplo, testes de segurança, testes de desempenho) exigem um ambiente de teste como o de produção para criar medidas realistas. Dependendo do tamanho e da complexidade do sistema testado, isto pode afetar muito o planejamento e o financiamento dos testes. Como o custo de tais ambientes pode ser alto, as seguintes alternativas podem ser levadas em consideração.

- A utilização do ambiente de produção;
- A utilização de uma versão reduzida do sistema.

Então, é preciso verificar se os resultados dos testes representam suficientemente o sistema de produção.

O momento da execução dos testes deve ser planejado atentamente e é bem provável que tais testes só possam ser executados em momentos específicos (por exemplo, em momentos de pouco uso).

4.2.4 Considerações organizacionais

Os testes técnicos podem envolver a medição do comportamento de diversos componentes em um sistema completo (por exemplo, servidores, bancos de dados, redes). Se tais componentes forem distribuídos em uma série de locais e organizações diferentes, os trabalhos necessários para planejar e coordenar os testes podem ser consideráveis. Por exemplo, certos componentes de *software* só podem ficar disponíveis para testes de sistema em momentos específicos do dia ou do ano ou as organizações só podem prestar suporte aos testes durante um número limitado de dias. A não confirmação de que os componentes do sistema e o pessoal (isto é, *expertise* “emprestada”) de outras organizações estão “de plantão” para fins de teste pode levar a uma

grave interrupção dos testes agendados.

4.2.5 Considerações de segurança de dados

É preciso levar em conta as medidas específicas de segurança que foram implementadas na etapa de planejamento de testes para garantir que todas as atividades de teste sejam viáveis. Por exemplo, a utilização da criptografia de dados pode dificultar a criação dos dados de teste e a confirmação dos resultados.

As políticas e as leis sobre a proteção da informação pode impedir a geração de quaisquer dados de teste necessários com base nos dados de produção. Anonimizar os dados de teste é uma tarefa não trivial que deve ser planejada nos termos da implementação de testes.

4.3 Teste de segurança

4.3.1 Introdução

O teste de segurança difere de outras formas de teste funcional de duas maneiras importantes:

1. As técnicas-padrão de seleção dos dados de entrada dos testes podem ignorar problemas de segurança importantes;
2. Os sintomas dos defeitos de segurança são muito diferentes dos encontrados em outros tipos de testes funcionais.

Os testes de segurança avaliam a vulnerabilidade de um sistema a ameaças através da tentativa de comprometer sua política de segurança. Segue uma lista de possíveis ameaças que devem ser exploradas durante o teste de segurança:

- Cópia não autorizada de aplicativos ou dados;
- Controle de acesso não autorizado (por exemplo, a capacidade de realizar tarefas para as quais o usuário não possui direitos); Os direitos, acessos e privilégios dos usuários são o foco deste teste. Estas informações devem estar disponíveis nas especificações no sistema;
- *Software* que exibe efeitos colaterais imprevistos quando realiza suas funções programadas. Por exemplo, um reprodutor de mídia que toca áudio corretamente, mas faz isto gravando arquivos em pastas temporárias sem criptografia, um efeito colateral que pode ser explorado por piratas de *software*;
- Código inserido em um *site* que pode ser utilizado pelos próximos usuários (*cross-site scripting* ou XSS, em inglês). O código pode ser malicioso;
- Transbordamento de dados (estouro de *buffer*), que pode ser provocado pela introdução de cadeias que superam a capacidade do código em um campo de entrada de interface de usuário. A vulnerabilidade do transbordamento de dados representa uma oportunidade para a execução de instruções de um código malicioso;
- Negação de serviço, que impede os usuários de interagir com um aplicativo (por exemplo, ao sobrecarregar um servidor da rede com solicitações “importunas”);
- A interceptação, a imitação e / ou a alteração e o consequente repasse de comunicados (por exemplo, transações de cartão de crédito) por parte de terceiros de modo que o usuário continue ignorando a presença dos terceiros (ataque *man-in-the-middle*);
- Decifração dos códigos de criptografia utilizados para proteger dados sensíveis;
- Bombas lógicas (conhecidas às vezes como ovos de Páscoa) que podem ser inseridas

maliciosamente no código e podem ser ativadas apenas em certas condições (por exemplo, em uma data específica). Quando as bombas lógicas são ativadas, podem realizar ações maliciosas, como a exclusão de arquivos ou a formatação de discos.

4.3.2 Planejamento de testes de segurança

Em geral, os seguintes aspectos são particularmente relevantes na hora de planejar os testes de segurança:

- Como os problemas de segurança podem ser introduzidos durante a elaboração da arquitetura, a concepção e a implementação do sistema, os testes de segurança podem ser agendados para os níveis de testes de unidade, de integração e de sistema. Devido às mudanças das ameaças à segurança, os testes de segurança também podem ser agendados com regularidade depois que o sistema tiver entrado em produção;
- As estratégias de teste propostas pelo *Technical Test Analyst* podem incluir revisões de códigos e análises estáticas com ferramentas de segurança. Elas podem ser eficazes na detecção de problemas de segurança na arquitetura, nos documentos de concepção e no código que são facilmente ignorados durante os testes dinâmicos;
- O *Technical Test Analyst* pode ser chamado a conceber e executar certos "ataques" contra a segurança (*vide* abaixo), que exigem muito planejamento e coordenação junto aos *stakeholders*. Outros testes de segurança podem ser realizados em parceria com os desenvolvedores ou com os *Test Analysts* (por exemplo, testes com os direitos, os acessos e os privilégios dos usuários). O planejamento dos testes de segurança deve incluir a consideração atenciosa de problemas organizacionais assim;
- Um aspecto essencial do planejamento de testes de segurança é a obtenção de aprovações. Para o *Technical Test Analyst*, isto significa a obtenção de uma autorização explícita do *Test Manager* para a realização dos testes de segurança planejados. Quaisquer outros testes não planejados que forem realizados podem parecer ataques reais e a pessoa que os realizar pode correr o risco de ser processada. Sem uma nota por escrito para provar a intenção e a autorização, talvez seja difícil explicar de maneira convincente a afirmação de que "era um teste de segurança";
- Vale lembrar que as melhorias que puderem ser feitas na segurança do sistema podem afetar seu desempenho. Depois de melhorar a segurança, é recomendável atender à necessidade de realizar testes de desempenho (*vide* o item 4.5 abaixo).

4.3.3 Especificações de testes de segurança

Certos testes de segurança podem ser agrupados [Whittaker04] de acordo com a origem do risco de segurança:

- *Relacionado à interface do usuário*: acesso não autorizado e entradas maliciosas;
- *Relacionado ao sistema de arquivos*: acesso aos dados sensíveis armazenados em arquivos ou repositórios;
- *Relacionado ao sistema operacional*: o armazenamento de informações sensíveis, como senhas sem criptografia, em memórias que podem ficar expostas quando o sistema entrar em pane por conta de entradas maliciosas;
- *Relacionado a software externo*: interações que podem ocorrer entre os componentes externos que o sistema utiliza. Podem vir da rede (por exemplo, pacotes ou mensagens incorretas repassadas) ou dos componentes do *software* (por exemplo, falha de um componente de que o *software* depende).

A seguinte abordagem [Whittaker04] pode ser utilizada para desenvolver testes de segurança:

- Reunir as informações que podem ser úteis para a especificação de testes, como nomes de funcionários, endereços físicos, detalhes das redes internas, números de IP, a identidade do *software* ou do *hardware* utilizado e versão do sistema operacional;
- Realizar uma varredura de vulnerabilidades com ferramentas amplamente disponíveis. Tais ferramentas não são utilizadas diretamente para comprometer o/s sistema/s, mas para identificar as vulnerabilidades que configuram uma infração da política de segurança ou podem levar a isso. As vulnerabilidades específicas também podem ser identificadas com *checklists* como as fornecidas pelo Instituto Nacional de Metrologia, Normalização e Qualidade Industrial dos EUA (NIST, na sigla em inglês) [Web-2];
- Desenvolver "planos de ataque" (isto é, um plano de ações de teste para comprometer a política de segurança de determinado sistema) com as informações coletadas. Várias entradas através de diversas interfaces (por exemplo, interface de usuário, sistema de arquivos) precisam ser definidas em planos de taque para detectar as falhas de segurança mais graves. Os diversos "ataques" descritos em [Whittaker04] são uma fonte valiosa de técnicas desenvolvidas especificamente para a realização de testes de segurança.

Os problemas de segurança também podem ser expostos pelas revisões (*vide* o capítulo 5) e / ou pela utilização de ferramentas de análise estática (*vide* o item 3.2). As ferramentas de análise estática contêm um grande conjunto de regras próprias das ameaças à segurança contra o qual o código é confrontado. Por exemplo, problemas de transbordamento de dados provocados pela não verificação do tamanho do *buffer* antes da atribuição de dados podem ser detectados pela ferramenta.

As ferramentas de análise estática podem ser utilizadas no código de rede para verificar uma possível exposição a vulnerabilidades de segurança, como injeções de código, *cookies* vulneráveis, *cross-site scripting*, adulterações de recursos e injeções de código SQL.

4.4 Teste de confiabilidade

Segundo a ISO 9126, a classificação das características de qualidade dos produtos define as seguintes subcaracterísticas de confiabilidade:

- Maturidade;
- Tolerância a falhas;
- Recuperabilidade.

4.4.1 Medição da maturidade do software

Os testes de confiabilidade procuram monitorar as estatísticas da maturidade do *software* com o passar do tempo e compará-las a uma meta desejada de confiabilidade, que pode ser expressa como nível de acordo de serviço. As medidas podem assumir a forma de tempo médio entre falhas (MTBF, na sigla em inglês), tempo médio até o restabelecimento (MTTR, na sigla em inglês) ou qualquer outra forma de medição de intensidade de falhas (por exemplo, o número de falhas de determinada gravidade de ocorrência semanal). Podem servir de critérios de saída (por exemplo, para o lançamento de produção).

4.4.2 Testes de tolerância a falhas

Além dos testes funcionais que avaliam a tolerância do *software* a falhas em termos de tratamento de valores

de entrada inesperados (os chamados testes negativos), outros testes são necessários para avaliar a tolerância a falhas de um sistema, que ocorrem fora do aplicativo testado. Tais falhas são normalmente divulgadas pelo sistema operacional (por exemplo, disco cheio, processo ou serviço indisponível, arquivo não encontrado, memória indisponível). Os testes de tolerância a falhas no sistema podem ser suportados por ferramentas específicas.

Os termos *robustez* e *tolerância a erros* também são muito utilizados quando a tolerância a falhas é discutida (*vide* [ISTQB_GLOSSARY] para obter mais detalhes).

4.4.3 Teste de recuperabilidade

Outras formas de testes de confiabilidade avaliam a capacidade que o sistema do *software* tem de se recuperar de falhas de *hardware* ou *software* de maneira predeterminada, o que, em seguida, permite a retomada das atividades normais. Entre os testes de recuperabilidade estão os testes sobre falhas e os de *backup* e restauração.

Os testes sobre falhas são realizados quando as consequências de uma falha no *software* forem tão negativas que medidas específicas referentes ao *hardware* e / ou ao *software* precisam ser implementadas para garantirem o funcionamento do sistema até mesmo em caso de falha. Os testes sobre falhas podem ser aplicados, por exemplo, quando o risco de perdas financeiras for extremo ou quando existirem problemas de segurança. Quando as falhas decorrerem de eventos catastróficos, esta forma de teste de recuperabilidade também pode se chamar teste de recuperação de desastres.

Normalmente, as medidas preventivas para falhas em *hardware* podem incluir o balanceamento de carga entre diversos processadores e a aglomeração de servidores, processadores ou discos para que um deles possa assumir se o outro falhar (sistemas redundantes). Normalmente, as medidas referentes ao *software* podem ser a implementação de mais de uma instância independente de um sistema de *software* (por exemplo, o sistema de controle de voo de uma aeronave) nos chamados sistemas redundantes dissimilares. Normalmente, os sistemas redundantes combinam medidas referentes ao *software* e ao *hardware* e podem ser sistemas duplos, triplos ou quádruplos, dependendo do número de instâncias independentes (duas, três ou quatro, respectivamente). O aspecto dissimilar do *software* é alcançado quando os mesmos requisitos do *software* são fornecidos a duas (ou mais) equipes de desenvolvimento independentes e não relacionadas com a finalidade de fazer os mesmos serviços serem fornecidos por um *software* diferente. Isto protege os sistemas redundantes dissimilares, já que uma entrada defeituosa semelhante tem menos chances de produzir os mesmos resultados. Estas medidas são tomadas para melhorar a recuperabilidade de um sistema e podem influenciar diretamente sua confiabilidade. Além disso, devem ser levadas em consideração ao realizar testes de confiabilidade.

O teste sobre falhas visa a testar sistemas explicitamente através da simulação de modos de falha ou da provocação de falhas em um ambiente controlado. Após uma falha, o mecanismo de *failover* é testado para garantir que os dados não sejam perdidos ou corrompidos e que os níveis de serviço acordados sejam mantidos (por exemplo, a disponibilidade de funções ou os tempos de resposta). Para obter mais informações referentes aos testes sobre falhas, *vide* [Web-1].

Os testes de *backup* e restauração se concentram em medidas de procedimento criadas para minimizar os efeitos de uma falha. Tais testes avaliam os procedimentos (normalmente documentados em um manual) por realizarem diferentes formas de *backup* e restauração dos dados em caso de perda ou corrupção de dados. Os casos de teste servem para garantir a cobertura de caminhos críticos ao longo de cada procedimento. As revisões técnicas podem realizar um *dry run* com estes cenários e confrontar os manuais com os

procedimentos reais. Os testes de aceitação operacional aplicam os cenários em um ambiente de produção ou como o de produção para validarem sua utilização real.

As medidas de testes de *backup* e restauração podem incluir:

- O tempo necessário para realizar tipos diferentes de *backup* (por exemplo, total, incremental);
- O tempo necessário para restaurar os dados;
- Os níveis de garantia de *backup* de dados (por exemplo, a recuperação de todos os dados de até 24 horas de existência, a recuperação de dados de transações específicas de até uma hora de existência).

4.4.4 Planejamento de testes de confiabilidade

Em geral, os seguintes aspectos são particularmente relevantes na hora de planejar os testes de confiabilidade:

- A confiabilidade pode continuar sendo monitorada depois que o *software* entrar em produção. A organização e o pessoal responsável pela operação do *software* devem ser consultados ao reunir os requisitos de confiabilidade para fins de planejamento de testes;
- O *Technical Test Analyst* pode selecionar um modelo de crescimento de confiabilidade que mostre os níveis esperados de confiabilidade com o tempo. O modelo de crescimento da confiabilidade pode proporcionar informações úteis ao *Test Manager* ao possibilitar a comparação dos níveis esperados e atingidos de confiabilidade;
- Os testes de confiabilidade devem ser realizados em um ambiente como o de produção. O ambiente utilizado deve continuar o mais estável possível para possibilitar o monitoramento das tendências de confiabilidade com o passar do tempo;
- Como frequentemente os testes de confiabilidade exigem a utilização do sistema inteiro, os testes de confiabilidade são normalmente realizados como parte dos testes de sistema. No entanto, componentes individuais podem ficar sujeitos a testes de confiabilidade e testes integrados de componentes. Revisões detalhadas da arquitetura, da modelagem e do código também podem ser utilizadas para eliminar parte do risco dos problemas de confiabilidade que ocorrem no sistema implementado;
- Para produzir resultados de testes estatisticamente significativos, os testes de confiabilidade normalmente exigem um tempo de execução longo. Isto pode dificultar o agendamento em relação a outros testes planejados.

4.4.5 Especificações de testes de confiabilidade

Os testes de confiabilidade podem assumir a forma de um conjunto repetido de testes predeterminados. Podem ser testes selecionados aleatoriamente de um *pool* ou casos de teste gerados por um modelo estatístico com métodos aleatórios ou pseudoaleatórios. Os testes também podem se basear em padrões de uso que, às vezes, são chamados de perfis operacionais (*vide* o item 4.5.4).

Certos testes de confiabilidade podem determinar a execução de ações que requerem muita memória várias vezes para que possíveis vazamentos de memória possam ser detectados.

4.5 Teste de desempenho

4.5.1 Introdução

Segundo a ISO 9126, a classificação das características de qualidade dos produtos inclui o desempenho (o comportamento ao longo do tempo) como subcaracterística de eficiência. Os testes de desempenho se concentram na capacidade de um componente ou de um sistema reagir às entradas de usuários ou do sistema dentro de um período específico e sob condições específicas.

As medições do desempenho variam de acordo com os objetivos do teste. Em relação a componentes de *software* individuais, o desempenho pode ser medido de acordo com os ciclos da CPU, ao passo que, em relação a sistemas baseados em clientes, o desempenho pode ser medido de acordo com o tempo necessário para responder uma solicitação específica de um usuário. Em sistemas cuja arquitetura consiste em diversos componentes (por exemplo, clientes, servidores, bancos de dados), o desempenho é medido nas transações entre componentes individuais para que os gargalos de desempenho possam ser identificados.

4.5.2 Tipos de teste de desempenho

4.5.1.1 Teste de carga

O teste de carga se concentra na capacidade de um sistema lidar com níveis cada vez maiores de cargas realistas antecipadas em decorrência de solicitações de transações geradas por usuários ou processos simultâneos. Os tempos médios de resposta referentes a usuários em cenários diferentes de uso típico (perfis operacionais) podem ser medidos e analisados. (*Vide* também [Splaine01].)

4.5.1.2 Teste de estresse

Os testes de estresse se concentram na capacidade que um sistema ou um componente tem de lidar com cargas máximas nos limites de suas cargas de trabalho antecipadas ou especificadas ou de ir além delas ou com uma disponibilidade reduzida de recursos, como capacidade acessível do computador e banda larga disponível. Os níveis de desempenho devem cair lenta e previsivelmente sem falhas à medida que os níveis de estresse aumentam. Em particular, a integridade funcional do sistema deve ser testada enquanto o sistema é submetido a estresse para detectar possíveis falhas no processamento funcional ou incoerências nos dados.

Um possível objetivo dos testes de estresse é o de descobrir os limites em que o sistema realmente falha para que o “elo mais fraco” possa ser determinado. Os testes de estresse permitem a adição de uma maior capacidade ao sistema em tempo hábil (por exemplo, memória, capacidade da CPU, armazenamento em banco de dados).

4.5.1.3 Teste de escalabilidade

Os testes de escalabilidade se concentram na capacidade que um sistema tem de atender a futuros requisitos de eficiências, que podem ir além dos exigidos atualmente. O objetivo dos testes é o de determinar a capacidade que o sistema tem de crescer (por exemplo, com mais usuários, quantidades maiores de dados armazenados) sem exceder os requisitos de desempenho atualmente especificados ou sem falhar. Quando os limites da escalabilidade forem conhecidos, os valores mínimos podem ser definidos e monitorados na produção para a geração de advertências referentes a problemas iminentes. Além disso, o ambiente de produção pode ser ajustado com um *hardware* adequado.

4.5.2 Planejamento de testes de desempenho

Além dos problemas gerais de planejamento descritos no item 4.2, os seguintes fatores podem influenciar o planejamento dos testes de desempenho:

- Dependendo do ambiente de teste utilizado e do *software* testado (*vide* o item 4.2.3), os testes de desempenho podem exigir a implementação do sistema inteiro antes da possibilidade de realizar testes eficazes. Neste caso, os testes de desempenho costumam ser agendados para que ocorram durante os testes de sistema. Outros testes de desempenho que podem ser realizados de maneira eficaz nos componentes podem ser agendados durante os testes de unidade;
- Em geral, é recomendável a realização de testes iniciais de desempenho o quanto antes, mesmo se o ambiente como o de produção ainda não estiver disponível. Os primeiros testes podem detectar problemas de desempenho (por exemplo, gargalos) e reduzir o risco do projeto ao evitar longas correções nas etapas posteriores do desenvolvimento ou da produção de *software*;
- As revisões do código, em particular as que se concentram na interação com o banco de dados, na interação com os componentes e no tratamento de erros, podem detectar problemas de desempenho (particularmente referentes à lógica do "aguardar e voltar a tentar" e a consultas ineficientes) e devem ser detectadas nos primeiros momentos do ciclo de vida do *software*;
- Devem existir planos e um orçamento para o *hardware*, o *software* e a banda larga de rede necessários para a execução dos testes de desempenho. As necessidades dependem, sobretudo, da carga que será gerada, o que pode se basear no número de usuários virtuais da simulação e no tráfego de rede que provavelmente será gerado. Sem isto, medições de desempenho não representativas podem ser realizadas. Por exemplo, a verificação dos requisitos de escalabilidade de um *site* muito acessado pode exigir a simulação de centenas de milhares de usuários virtuais;
- A geração da carga necessária para os testes de desempenho pode afetar bastante os custos de aquisição de *hardware* e ferramentas. Isto deve ser levado em consideração no planejamento dos testes de desempenho para garantir a disponibilidade de verbas suficientes;
- Os custos da geração da carga nos testes de desempenho podem ser minimizados através do aluguel da infraestrutura de teste. Por exemplo, isto pode envolver o aluguel de licenças totais de ferramentas de desempenho ou o uso de serviços de prestadores terceirizados para atender às necessidades de *hardware* (por exemplo, serviços na nuvem). Se esta abordagem for realizada, o tempo disponível para a realização dos testes de desempenho pode ser limitado e, assim, deve ser cuidadosamente planejado;
- Na etapa de planejamento, é preciso garantir que a ferramenta de desempenho que será utilizada gere a compatibilidade necessária com os protocolos de comunicação utilizados pelo sistema testado;
- Os defeitos relacionados ao desempenho frequentemente causam um impacto considerável no sistema testado. Quando os requisitos de desempenho forem obrigatórios, frequentemente, realizar testes de desempenho com componentes cruciais (com controladores e simuladores) é mais útil do que aguardar os testes de sistema.

4.5.3 Especificações de testes de desempenho

A especificação dos testes para tipos diferentes de testes de desempenho, como os testes de carga e de estresse, se baseia na definição dos perfis operacionais. Eles representam formas diferentes de comportamento do usuário na interação com o aplicativo. Podem existir vários perfis operacionais em determinado aplicativo.

O número de usuários por perfil operacional pode ser obtido com monitoramento das ferramentas (quando o

aplicativo real ou semelhante já estiver disponível) ou com a projeção do uso. Tais previsões podem se basear em algoritmos ou podem ser fornecidas pela organização de negócios e são especialmente importantes na especificação dos perfis operacionais que serão utilizados nos testes de escalabilidade.

Os perfis operacionais servem de base para o número e os tipos de casos de teste que serão utilizados durante os testes de desempenho. Os testes são frequentemente controlados por ferramentas de teste que criam usuários "virtuais" ou simulados em quantidades que representam o perfil testado (*vide* o item 6.3.2).

4.6 Utilização de recursos

Segundo a ISO 9126, a classificação das características de qualidade dos produtos inclui a utilização de recursos como subcaracterística de eficiência. Os testes que dizem respeito à utilização de recursos comparam o uso dos recursos do sistema (por exemplo, o uso de memória, da capacidade do disco, da banda larga da rede, das conexões) com um *benchmark* predefinido. O uso é comparado quando exposto a cargas normais e em situações de estresse, como altos níveis de transações e volume de dados, para determinar se há um crescimento antinatural do uso.

Por exemplo, em sistemas incorporados em tempo real, a utilização da memória (às vezes denominada pegada de memória) desempenho um papel importante nos testes de desempenho. Se a pegada de memória exceder a medida permitida, o sistema pode ter memória insuficiente para a realização das tarefas dentro de períodos específicos. Isto pode deixar o sistema lento ou até mesmo travá-lo.

A análise dinâmica também pode ser aplicada à investigação da utilização de recursos (*vide* o item 3.3.4) e à detecção de gargalos de desempenho.

4.7 Teste de manutenibilidade

Frequentemente, ao longo de seu ciclo de vida, o *software* passa muito mais tempo sendo mantido do que desenvolvido. Os testes de manutenção são realizados para testar as alterações em um sistema operacional ou o impacto de um ambiente alterado sobre um sistema operacional. Para ter a certeza de que a manutenção é a mais eficiente possível, os testes de manutenibilidade são realizados para a medir a facilidade com a qual o código pode ser analisado, alterado e testado.

Normalmente, entre os objetivos de manutenibilidade dos *stakeholders* afetados (por exemplo, o proprietário ou o operador do *software*) estão:

- A minimização dos custos de possuir ou operar o *software*;
- A minimização do tempo de inatividade necessário para a manutenção do *software*.

Os testes de manutenibilidade devem ser incluídos na estratégia e / ou na abordagem de teste na presença de um ou mais dos seguintes fatores:

- As alterações no *software* são prováveis depois que o *software* entra em produção (por exemplo, para corrigir defeitos e lançar atualizações planejadas);
- As vantagens de atingir os objetivos de manutenibilidade (*vide supra*) ao longo do ciclo de vida do *software* são levadas em consideração pelos *stakeholders* afetados para superar os custos da realização de testes de manutenibilidade e fazer as alterações necessárias;
- Os riscos de uma manutenibilidade ruim no *software* (por exemplo, longos tempos de resposta a defeitos comunicados pelos usuários e / ou pelos clientes) justificam a realização de testes de

manutenibilidade.

As técnicas adequadas dos testes de manutenibilidade incluem a análise estática e as revisões discutidas nos itens 3.2 e 5.2. Os testes de manutenibilidade devem ser iniciados assim que os documentos de modelagem estiverem disponíveis e devem continuar enquanto os trabalhos de implementação do código durarem. Como a manutenibilidade foi incorporada ao código e cada componente individual do código foi documentado, a manutenibilidade pode ser avaliada no início do ciclo de vida sem a necessidade de aguardar a conclusão do sistema.

Os testes de manutenibilidade dinâmicos se concentram em procedimentos documentados e desenvolvidos para a manutenção de um aplicativo específico (por exemplo, atualizações de *software*). Os cenários de manutenção escolhidos servem de casos de teste para garantir que os níveis de serviço exigidos sejam atingidos com os procedimentos documentados. Esta forma de teste é particularmente relevante quando a infraestrutura subjacente for complexa e os procedimentos de apoio podem envolver muitos departamentos / muitas organizações. Esta forma de teste pode acontecer como parte dos testes de aceitação operacional. [Web-1]

4.7.1 Analisabilidade, modificabilidade, estabilidade e testabilidade

A manutenibilidade de um sistema pode ser medida em termos dos trabalhos necessários para diagnosticar os problemas identificados em um sistema (analisabilidade), implementar as alterações nos códigos (modificabilidade) e testar o sistema alterado (testabilidade). A estabilidade diz respeito especificamente à resposta do sistema à mudança. Os sistemas com baixa estabilidade exibem grandes números de problemas a jusante (também conhecidos como efeito dominó) quando uma alteração é feita. [ISO9126] [Web-1]

Os trabalhos necessários para a realização de tarefas de manutenção dependem de uma série de fatores, como a metodologia de concepção de *software* (por exemplo, a orientação a objetos) e as normas de codificação utilizadas.

Neste contexto, estabilidade não deve ser confundida com robustez ou tolerância a falhas, que são discutidas no item 4.4.2.

4.8 Teste de portabilidade

Em geral, os testes de portabilidade dizem respeito à facilidade com a qual o *software* pode ser transferido a seu ambiente pretendido, quer inicialmente, quer a partir de um ambiente existente. Os testes de portabilidade incluem os testes de instalabilidade, coexistência / compatibilidade, adaptabilidade e substitutibilidade. Os testes de portabilidade podem começar por componentes individuais (por exemplo, a modificabilidade de um componente específico, como a mudança de um sistema de gerenciamento de bancos de dados para outro) e seu escopo cresce com a disponibilidade do código. Às vezes, a instalabilidade só pode ser testada quando todos os componentes do produto estiverem funcionando. A portabilidade deve ser concebida e incorporada ao produto e, assim, deve ser considerada nos primeiros momentos das fases de concepção e arquitetura. As revisões de arquitetura e concepção podem ser particularmente produtivas na hora de identificar os possíveis requisitos e problemas de portabilidade (por exemplo, a dependência de um sistema operacional específico).

4.8.1 Teste de instalabilidade

Os testes de instalabilidade são realizados no *software* e nos procedimentos escritos que são utilizados para

instalar o *software* no ambiente-alvo. Por exemplo, isto pode incluir o *software* desenvolvido para a instalação de um sistema operacional em um processador ou um assistente para a instalação de um produto no computador de um cliente.

Entre os objetivos comuns dos testes de instalabilidade estão:

- A validação de que o *software* consegue ser instalado com sucesso ao seguir as instruções em um manual de instalação (inclusive a execução de qualquer *script* de instalação) ou ao utilizar um assistente de instalação. Isto inclui as opções de instalação de configurações diferentes de *hardware* / *software* e diversos graus de instalação (por exemplo, instalação inicial ou atualização);
- A realização de testes para saber se as falhas que ocorrem durante a instalação (por exemplo, o não carregamento de DLLs específicos) são tratadas corretamente pelo *software* de instalação sem deixar o sistema em um estado indefinido (por exemplo, com o *software* parcialmente instalado ou com as configurações incorretas no sistema);
- A realização de testes para saber se é possível concluir uma instalação / desinstalação parcial;
- A realização de testes para saber se o assistente de instalação consegue identificar com sucesso plataformas de *hardware* ou configurações inválidas no sistema operacional;
- A verificação de que o processo de instalação pode ser concluído em um número específico de minutos ou em menos do que certo número de etapas;
- A validação de que é possível fazer um *downgrade* no *software* ou desinstalá-lo.

Normalmente, o teste de funcionalidade é realizado após o teste de instalação para a detecção de quaisquer falhas que possam ter sido introduzidas pela instalação (por exemplo, configurações incorretas, funções indisponíveis). Normalmente, o teste de usabilidade é realizado junto com o teste de instalabilidade (por exemplo, para verificar se os usuários receberam instruções compreensíveis e *feedback* / mensagens de erro durante a instalação).

4.8.2 Teste de coexistência / compatibilidade

Os sistemas de informática que não estiverem relacionados são considerados compatíveis quando conseguir funcionar no mesmo ambiente (por exemplo, no mesmo *hardware*) sem afetar o comportamento um do outro (por exemplo, conflitos de recursos). O teste de compatibilidade deve ser realizado quando um *software* novo ou um *upgrade* for lançado em ambientes que já contêm os aplicativos instalados.

Problemas de compatibilidade podem surgir quando o aplicativo for testado em um ambiente em que é o único aplicativo instalado (quando os problemas de incompatibilidade não são detectáveis) e, então, implantado em outro ambiente (por exemplo, o de produção), que também executa outros aplicativos.

Entre os objetivos comuns dos testes de compatibilidade estão:

- A avaliação de possíveis impactos adversos sobre a funcionalidade quando os aplicativos são carregados no mesmo ambiente (por exemplo, o uso conflitante de recursos quando um servidor executa vários aplicativos);
- A avaliação do impacto sobre qualquer aplicativo decorrente da implantação de correções e *upgrades* no sistema operacional.

Os problemas de compatibilidade devem ser analisados ao planejar o ambiente de produção objetivado, mas os testes reais são normalmente realizados após a conclusão bem-sucedida dos testes de sistema e de aceitação do usuário.

4.8.3 Teste de adaptabilidade

Os testes de adaptabilidade verificam se determinado aplicativo consegue funcionar corretamente em todos os ambientes-alvo (*hardware*, *software*, *middleware*, sistema operacional *etc.*). Portanto, um sistema adaptável é um sistema aberto que consegue ajustar seu comportamento de acordo com as mudanças no ambiente ou nas peças do próprio sistema. A especificação de testes de adaptabilidade exige que as combinações dos ambientes-alvo sejam identificadas, configuradas e disponibilizadas à equipe de teste. Então, os ambientes são testados com uma seleção de casos de teste funcionais que utilizam os diversos componentes presentes no ambiente.

A adaptabilidade pode dizer respeito à capacidade que o *software* tem de ser adaptado a diversos ambientes específicos através da realização de um procedimento predefinido. Os testes podem avaliar este procedimento.

Os testes de adaptabilidade podem ser realizados junto com os testes de instalabilidade e, normalmente, são seguidos pelos testes funcionais para a detecção de qualquer falha que possa ter sido introduzida na adaptação do *software* a um ambiente diferente.

4.8.4 Teste de substitutibilidade

Os testes de substitutibilidade se concentram na capacidade que os componentes do *software* em um sistema têm de ser trocados por outros. Isto pode ser particularmente relevante para os sistemas que utilizam *software* comercial de prateleira (*commercial off-the-shelf* ou COTS, na sigla em inglês) para componentes de sistemas específicos.

Os testes de substitutibilidade podem ser realizados junto com os testes de integração funcional quando mais de um componente alternativo estiver disponível para integração ao sistema inteiro.

A substitutibilidade pode ser avaliada por uma revisão ou uma inspeção técnica nos níveis de arquitetura e modelagem, quando a definição clara das interfaces com possíveis componentes substituíveis é enfatizada.

5 Revisões – 165 minutos

Palavras-chave

antipadrão

Objetivos de aprendizagem das revisões

5.1 Introdução

TTA 5.1.1 (K2) Explicar por que os preparativos das revisões são importantes para o *Technical Test Analyst*.

5.2 Utilização de checklists em revisões

TTA 5.2.1 (K4) Analisar um projeto de arquitetura e identificar problemas de acordo com a *checklist* fornecida no *syllabus*;

TTA 5.2.2 (K4) Analisar uma parte do código ou do pseudocódigo e identificar problemas de acordo com uma a *checklist* fornecida no *syllabus*.

5.1 Introdução

O *Technical Test Analyst* deve participar ativamente do processo de revisão e oferecer seus pontos de vista únicos. O *Technical Test Analyst* deve contar com treinamento formal em revisão para compreender melhor suas funções respectivas em qualquer processo de revisão técnica. Todos os participantes de revisões devem estar comprometidos com os benefícios de uma revisão técnica bem feita. (Para uma descrição completa das revisões técnicas, inclusive diversas *checklists* de revisão, *vide* [Wiegers02]). Normalmente, o *Technical Test Analyst* participa de revisões e inspeções técnicas, às quais oferece um ponto de vista operacional (comportamental) que pode fazer falta aos desenvolvedores. Além disso, o *Technical Test Analyst* desempenha um papel importante na definição, na aplicação e na manutenção das informações das *checklists* de revisão e da gravidade dos defeitos.

Independentemente do tipo de revisão realizada, o *Technical Test Analyst* deve reservar tempo o suficiente para os preparativos. Isto inclui tempo para revisar o produto de trabalho, verificar os documentos cruzados para comprovar sua coerência e determinar o que pode estar faltando no produto de trabalho. Sem tempo suficiente para os preparativos, a revisão pode virar uma edição em vez de ser uma verdadeira revisão. Uma boa revisão inclui a compreensão do que está escrito, a determinação do que está faltando e a comprovação de que o produto descrito mantém a coerência com outros produtos que ou já foram desenvolvidos ou estão em desenvolvimento. Por exemplo, ao revisar um plano de teste de integração, o *Technical Test Analyst* também deve levar em consideração os itens que foram integrados. Estão prontos para a integração? Existem dependências que precisam ser documentadas? Existem dados disponíveis para testar os pontos da integração? As revisões não independem do produto de trabalho revisado. Também deve levar em consideração a interação daquele item com os outros no sistema.

Não é incomum que o autor de um produto revisado se sinta criticado. O *Technical Test Analyst* deve fazer de tudo para abordar quaisquer comentários sobre a revisão partindo da premissa de colaboração com o autor para a criação do melhor produto possível. Ao usar esta abordagem, os comentários serão feitos de maneira construtiva e ficarão voltados para o produto de trabalho e não para o autor. Por exemplo, se uma declaração for ambígua, é melhor afirmar "Não sei o que devo testar para comprovar se este requisito foi implementado corretamente. Pode me ajudar a entender isto?" do que "Este requisito é ambíguo e ninguém conseguirá decifrá-lo".

Em uma revisão, o trabalho do *Technical Test Analyst* consiste em garantir que as informações fornecidas no produto de trabalho sejam suficientes para fundamentar o teste. Se as informações inexistirem ou não forem claras, então, provavelmente, isto é um defeito que precisa ser corrigido pelo autor. Ao manter uma abordagem positiva em vez de uma abordagem crítica, os comentários serão mais bem recebidos e a reunião será mais produtiva.

5.2 Utilização de checklists em revisões

As *checklists* são usadas durante as revisões para lembrar os participantes de que devem verificar pontos específicos durante a revisão. Além disso, as *checklists* podem ajudar a despersonalizar a revisão, por exemplo: "É a mesma *checklist* que usamos em todas as revisões e não só em seu produto de trabalho". As *checklists* podem ser genéricas e são utilizadas em todas as revisões ou podem se focar em características ou áreas específicas de qualidade. Por exemplo, uma *checklist* genérica poderia verificar o bom uso de termos como *deverá* e *deve*, a formatação adequada e itens de conformidade parecidos. Uma *checklist* direcionada

pode se concentrar em problemas de segurança ou de desempenho.

As *checklists* mais úteis são as desenvolvidas gradualmente por uma organização individual, pois refletem:

- A natureza do produto;
- O ambiente de desenvolvimento local:
  - Pessoal;
  - Ferramentas;
  - Prioridades.
- O histórico de sucessos e defeitos anteriores;
- Problemas específicos (por exemplo, desempenho e segurança).

As *checklists* devem ser adaptadas à organização e, talvez, ao projeto específico. As *checklists* constantes neste capítulo devem servir apenas de exemplo.

Algumas organizações ampliam o conceito comum de *checklist* de *software* e incluem *antipadrões* que dizem respeito a erros comuns, técnicas ruins e outras práticas ineficazes. O termo provém do conceito popular de padrão de desenho, que é uma solução reutilizável para problemas comuns que demonstrou ser eficaz na prática [Gamma94]. Assim, o antipadrão é um erro cometido muitas vezes e frequentemente implementado como atalho conveniente.

Vale lembrar que se um requisito não for testável, o que significa que foi definido de maneira que o *Technical Test Analyst* não consegue nem determinar como testá-lo, há um defeito nele. Por exemplo, um requisito que declarar "O *software* deve ser rápido" não é testável. Como o *Technical Test Analyst* pode determinar se o *software* é rápido? Se, pelo contrário, o requisito disser "O *software* deve gerar um tempo máximo de resposta de três segundos em condições de carga específicas", a testabilidade do requisito é muito melhor se as "condições de carga específicas" forem definidas (por exemplo, o número de usuários simultâneos, as atividades realizadas pelos usuários). Também se trata de um requisito geral, pois pode facilmente gerar muitos casos de teste individuais em um aplicativo não trivial. A rastreabilidade a partir deste requisito até os casos de teste também é crucial porque se a especificação requisito mudar, todos os casos de teste precisarão ser revisados e atualizados quando necessário.

5.2.1 Revisões de arquiteturas

A arquitetura do *software* consiste na organização fundamental de um sistema incorporada a seus componentes, às suas inter-relações, às suas relações com o ambiente e aos princípios que regem sua concepção e sua evolução. (*Vide* [ANSI/IEEE Std 1471-2000] e [Bass03].)

As *checklists* utilizadas nas revisões de arquitetura podem, por exemplo, incluir a verificação da implementação adequada dos seguintes itens, que são mencionados em [Web-3]:

- *Pool* de conexão: a redução dos custos do tempo de execução ligados à definição de conexões com um banco de dados através do estabelecimento de um *pool* comum de conexões;
- Balanceamento de carga: a distribuição equitativa da carga entre uma série de recursos;
- Processamento distribuído;
- *Caching*: a utilização de uma cópia local de dados para a redução do tempo de acesso;
- Inicialização lenta;
- Simultaneidade de transações;
- Isolamento de processos entre o processamento de transações em tempo real (OLTP, na sigla em inglês) e o processamento analítico em tempo real (OLAP, na sigla em inglês);

- Replicação de dados.

Outros detalhes (não relevantes para o exame de certificação) podem ser encontrados em [Web-4], que diz respeito a um estudo de 117 *checklists* de 24 fontes. As diferentes categorias de itens das *checklists* são discutidas e exemplos de bons e maus itens de *checklist* são fornecidos.

5.2.2 Revisões de códigos

Em relação às revisões de códigos, as *checklists* são necessariamente muito detalhadas e, assim como as *checklists* referentes às revisões de arquiteturas, são mais úteis quando tratam especificamente de uma linguagem, de um projeto e de uma empresa. A inclusão de antipadrões de código é útil, particularmente para os desenvolvedores de *software* menos experientes.

As *checklists* utilizadas em revisões de códigos podem incluir os seis itens seguintes (com base em [Web-5]):

1. Estrutura:

- O código implementa o projeto total e corretamente?
- O código observa as normas de codificação pertinentes?
- O código é bem estruturado, possui coerência de estilo e conta com uma formatação coerente?
- Existem procedimentos desnecessários ou códigos inalcançáveis?
- Existem restos de simuladores ou rotinas de teste no código?
- Qualquer código pode ser substituído por chamadas de componentes reutilizáveis externos ou bibliotecas de funções?
- Existem blocos de código repetido que podem ser reunidos em um único procedimento?
- O armazenamento é eficiente?
- Os símbolos são mais utilizados do que constantes de “números mágicos” ou cadeias?
- Os módulos são excessivamente complexos e devem ser reestruturados ou divididos em vários módulos?

2. Documentação:

- O código foi clara e suficientemente documentado com um estilo de comentário fácil de manter?
- Todos os comentários conferem com o código?
- A documentação observa as normas pertinentes?

3. Variáveis:

- Todas as variáveis foram definidas adequadamente com nomes significativos, coerentes e claros?
- Existem variáveis redundantes ou sem uso?

4. Operações aritméticas:

- O código evita a comparação da igualdade de números em ponto flutuante?
- O código impede erros de arredondamento sistematicamente?
- O código evita somas e subtrações de números de magnitudes muito diferentes?
- Os divisores são testados em caso de zero ou ruído?

5. *Loops* e desvios:

- Todos os *loops*, desvios e construtos lógicos estão completos, corretos e adequadamente

posicionados?
- Os casos mais comuns são testados primeiro em cadeias SE-SENÃOSE?
- Todos os casos são cobertos em um bloco SE-SENÃOSE ou CASO, inclusive as cláusulas SENÃO ou PADRÃO?
- Cada comando de caso possui um padrão?
- As condições de terminação em *loop* são óbvias e invariavelmente alcançáveis?
- Os índices ou os subscritos são inicializados adequadamente antes do *loop*?
- Os comandos dentro dos *loops* podem ser colocados fora dos *loops*?
- O código no *loop* evita a manipulação da variável do índice ou sua utilização após a saída do *loop*?

6. Programação defensiva:

- Os limites de arranjos, registros ou arquivos de índices, ponteiros e subscritos foram testados?
- A validade e a integridade dos argumentos de dados e entradas importados foram testadas?
- Todas as variáveis de saída foram atribuídas?
- O elemento de dados correto foi utilizado em cada comando?
- Todas as alocações de memória foram liberadas?
- Os tempos de espera excedidos ou as armadilhas de erro são utilizados para o acesso a dispositivos externos?
- A existência dos arquivos é verificada antes de tentar acessá-los?
- Os arquivos e os dispositivos são deixados no estado correto após o fechamento do programa?

Para obter outros exemplos de *checklists* utilizadas em revisões de códigos em níveis diferentes de teste, *vide* [Web-6].

6 Ferramentas e automação de testes – 195 minutos

Palavras-chave

teste orientado a dados, ferramenta de depuração, ferramenta de semeamento de falhas, ferramenta de teste de *hyperlink*, teste orientado a palavras-chave, ferramenta de teste de desempenho, ferramenta de registro / execução, analisador estático, ferramenta de execução de testes, ferramenta de gerenciamento de testes

Objetivos de aprendizagem das ferramentas e automação de testes

6.1 Integração e troca de informações entre ferramentas

TTA-6.1.1 (K2) Explicar os aspectos técnicos que devem ser levados em consideração quando várias ferramentas são utilizadas ao mesmo tempo.

6.2 Definição do projeto de automação de testes

TTA-6.2.1 (K2) Resumir as atividades que o *Technical Test Analyst* realiza ao definir um projeto de automação de testes;

TTA-6.2.2 (K2) Resumir as diferenças entre a automação orientada a dados e a orientada a palavras-chave;

TTA-6.2.3 (K2) Resumir os problemas técnicos comuns que fazem que os projetos de automação não alcancem o retorno sobre o investimento planejado;

TTA-6.2.4 (K3) Criar uma tabela de palavras-chave com base em determinado processo de negócios.

6.3 Ferramentas de teste específicas

TTA-6.3.1 (K2) Resumir a finalidade das ferramentas de semeamento e injeção de falhas;

TTA-6.3.2 (K2) Resumir as principais características e os problemas de implementação das ferramentas de teste e monitoramento de desempenho;

TTA-6.3.3 (K2) Explicar a finalidade geral das ferramentas utilizadas nos testes baseados na rede;

TTA-6.3.4 (K2) Explicar como as ferramentas fundamentam o conceito de teste baseado em modelos;

TTA-6.3.5 (K2) Expor a finalidade das ferramentas utilizadas para fundamentar o processo de teste de componentes e de pacotes.

6.1 Integração e troca de informações entre ferramentas

Embora o *Test Manager* seja responsável pela seleção e pela integração das ferramentas, o *Technical Test Analyst* pode ter que revisar a integração de uma ferramenta ou um conjunto de ferramentas para garantir o rastreamento preciso dos dados decorrentes de diversas áreas de teste, como análise estática, automação de execução de testes e gerenciamento de configurações. Além disso, dependendo das habilidades de programação do *Technical Test Analyst*, talvez tenha que participar da criação do código que integrará as ferramentas que não se integram "fora da máquina".

Um conjunto de ferramentas ideal deve eliminar a duplicidade das informações nas ferramentas. Armazenar os *scripts* da execução de testes em um banco de dados de gerenciamento de testes e no sistema de gerenciamento de configurações dá mais trabalho e a possibilidade de erros é maior. Seria melhor contar com um sistema de gerenciamento de testes que incluísse um componente de gerenciamento de configurações ou realizasse a integração com uma ferramenta de gerenciamento de configurações já existente na organização. A boa integração das ferramentas de rastreamento de defeitos e de gerenciamento de testes permitirá que o testador lance um relatório de defeitos durante a execução dos casos de teste sem ter que encerrar a ferramenta de gerenciamento de testes. A boa integração das ferramentas de análise estática deve conseguir comunicar quaisquer incidentes e avisos descobertos ao sistema de gerenciamento de defeitos (embora isto seja configurável devido aos muitos avisos que podem ser gerados).

A aquisição de uma suíte de ferramentas de teste do mesmo fornecedor não significa automaticamente que as ferramentas funcionarão juntas de maneira adequada. Ao considerar a abordagem à integração de ferramentas, o foco datacêntrico é preferível. Os dados devem ser trocados sem intervenção manual em tempo hábil com acurácia garantida, inclusive recuperação de falhas. Embora seja útil que o usuário tenha uma experiência coerente, a captura, o armazenamento, a proteção e a apresentação dos dados devem ser o foco principal da integração das ferramentas.

A organização deve avaliar o custo da automatização do intercâmbio de informações em comparação com o risco de perder informações ou permitir que os dados saiam de sincronia devido à necessidade de intervenção manual. Como a integração pode ser cara ou difícil, deve ser uma consideração importante na estratégia geral de ferramentas.

Alguns ambientes de desenvolvimento integrado podem simplificar a integração entre as ferramentas que conseguem funcionar em tal ambiente. Isto facilita a unificação do *look and feel* das ferramentas que compartilham a mesma estrutura. No entanto, uma interface de usuário parecida não garantirá uma troca tranquila de informações entre os componentes. Talvez haja necessidade de codificação para concluir a integração.

6.2 Definição do projeto de automação de testes

Para serem rentáveis, as ferramentas de teste e, particularmente, de automação de testes devem ser cuidadosamente arquitetadas e projetadas. A implementação de uma estratégia de automação de testes sem uma arquitetura sólida costuma resultar em um conjunto de ferramentas que é caro de se manter, insuficiente para a finalidade e incapaz de alcançar o retorno sobre o investimento almejado.

Os projetos de automação de testes devem ser considerados projetos de desenvolvimento de *software*. Isto inclui a necessidade de documentos de arquitetura, documentos detalhados de projeto, revisões de projeto e

código, componentes e testes de integração de componentes e testes finais de sistema. Os testes podem ser desnecessariamente atrasados ou complicados quando um código instável ou inexato de automação de testes for utilizado. Existem várias atividades que o *Technical Test Analyst* realiza em relação à automação de testes. Entre elas estão:

- A determinação do responsável pela execução dos testes;
- A seleção da ferramenta adequada para a organização, o prazo, a habilidade da equipe, os requisitos de manutenção (talvez isto implique a criação de uma ferramenta em vez da aquisição de uma);
- A definição dos requisitos de interface entre a ferramenta de automação e outras ferramentas como as de gerenciamento de testes e de defeitos;
- A seleção de uma abordagem à automação, isto é, orientada a palavras-chave ou dados (*vide* o item 6.2.1 abaixo);
- A colaboração com o *Test Manager* para estimar o custo da implementação, inclusive a capacitação;
- O agendamento do projeto de automação e a alocação do tempo para manutenção;
- A capacitação de *Test Analysts* e *Business Analysts* na utilização e no fornecimento de dados para a automação;
- A determinação do método de execução dos testes automatizados;
- A determinação do método de combinação dos resultados dos testes automatizados com os resultados dos testes manuais.

Estas atividades e as decisões resultantes influenciarão a escalabilidade e a manutenibilidade da solução de automação. É preciso passar tempo o suficiente pesquisando as opções, investigando as ferramentas e as tecnologias disponíveis e compreendendo os planos futuros da organização. Algumas destas atividades exigem mais consideração do que outras, particularmente durante o processo decisório. Elas são discutidas mais detalhadamente nas seções seguintes.

6.2.1 Seleção da abordagem de automação

A automação de testes não se limita aos testes através da interface gráfica de usuário. As ferramentas existem para facilitar a automação dos testes no nível API através de uma interface de linha de comando (CLI, na sigla em inglês) e outros pontos de interface no *software* testado. Uma das primeiras decisões que o *Technical Test Analyst* deve tomar é a interface mais eficaz que deve ser acessada para automatizar os testes.

Uma das dificuldades de realizar testes através da interface gráfica de usuário é a tendência que ela tem de mudar à medida que o *software* evolui. Dependendo de como o código de automação de testes foi projetado, isto pode resultar em um fardo considerável para a manutenção. Por exemplo, a utilização do recurso de registro e execução em uma ferramenta de automação de testes pode levar a casos de teste automatizados (frequentemente chamados de *scripts* de teste) que não são mais executados da maneira desejada se a interface gráfica de usuário mudar. Isto acontece porque o *script* registrado captura as interações com os objetos gráficos quando o testador executa o *software* manualmente. Se os objetos acessados mudarem, os *scripts* registrados talvez precisem ser atualizados também para refletirem tais alterações.

As ferramentas de captura e execução podem ser um bom ponto de partida para o desenvolvimento de *scripts* de automação. O testador registra a sessão de teste e o *script* registrado é modificado para aprimorar a manutenibilidade (por exemplo, através da substituição de seções por funções reutilizáveis no *script* registrado).

Dependendo do *software* testado, os dados utilizados em cada teste podem ser diferentes, embora as etapas executadas dos testes sejam praticamente idênticas (por exemplo, o tratamento de erros de teste em um

campo de entrada pela digitação de vários valores inválidos e da verificação dos retornos de erro referentes a cada um deles). São ineficientes o desenvolvimento e a manutenção de *scripts* de teste automatizados para cada um dos valores que serão testados. Uma solução técnica comum para este problema consiste na transferência dos dados dos *scripts* para um unidade externa de armazenamento, como uma planilha ou um banco de dados. As funções são elaboradas para permitirem o acesso a dados específicos de cada execução do *script* de teste, o que possibilita que um único *script* funcione através de um conjunto de dados de teste que fornecer os valores de entrada e os valores dos resultados esperados (por exemplo, um valor indicado em um campo de texto ou uma mensagem de erro). Esta abordagem é orientada a dados. Com esta abordagem, o *script* de teste que processará os dados fornecidos é desenvolvido, assim como o ambiente e a infraestrutura necessárias para apoiar a execução do *script* ou de um conjunto de *scripts*. Na realidade, os dados constantes na planilha ou no banco de dados são criados pelo *Test Analyst*, que conhece a função de negócios do *software*. A divisão do trabalho permite que os responsáveis pelo desenvolvimento de *scripts* de teste (por exemplo, o *Technical Test Analyst*) se concentrem na implementação de *scripts* de automação inteligentes enquanto o *Test Analyst* mantém a responsabilidade pelo teste real. Na maioria dos casos, o *Test Analyst* será responsável pela execução de *scripts* de teste quando a automação for implementada e testada.

Outra abordagem, a orientada a palavras-chave ou comandos, vai além com a separação da ação a ser realizada nos dados fornecidos do *script* de teste [Buwalda01]. Para conseguir esta separação, uma metalinguagem de alto nível mais descritiva do que diretamente executável é criada por especialistas em domínios (por exemplo, o *Test Analyst*). Cada comando da linguagem descreve um processo de negócios total ou parcial do domínio que pode precisar de testes.

6.2.2 Modelagem de processos de negócios de automação

A fim de implementar uma abordagem orientada a palavras-chave na automação de testes, os processos de negócios que serão testados devem ser modelados na linguagem de palavras-chave de alto nível. É importante que a linguagem seja intuitiva para os usuários, os quais provavelmente serão os *Test Analysts* que atuam no projeto.

Geralmente, as palavras-chave são utilizadas para representar as interações de negócios de alto nível com o sistema. Por exemplo, *Cancelar_Pedido* pode exigir a verificação da existência do pedido, a confirmação dos direitos de acesso da pessoa que solicitou o cancelamento, a exibição do pedido que será cancelado e a solicitação de uma confirmação do cancelamento. As sequências de palavras-chave (por exemplo, *Login*, *Selecionar_Pedido*, *Cancelar_Pedido*) e os dados dos testes relevantes são utilizados pelo *Test Analyst* para especificar os casos de teste. Segue uma tabela simples de entradas orientadas a palavras-chave que pode ser utilizada para testar a capacidade que o *software* tem de acrescentar, zerar e excluir contas de usuário:

Palavra-chave	***Usuário***	***Senha***	***Resultado***
Adicionar_Usuario	Usuario1	Senha1	Mensagem adicionada por usuário
Adicionar_Usuario	@Rec34	@Rec35	Mensagem adicionada por usuário
Trocar_Senha	Usuario1	Bem-vindo	Mensagem de confirmação de troca de senha
Excluir_Usuario	Usuario1		Mensagem de usuário / senha inválida
Adicionar_Usuario	Usuario3	Senha3	Mensagem adicionada por usuário
Excluir_Usuario	Usuario2		Mensagem de usuário não encontrado

O *script* de automação que utiliza esta tabela buscaria os valores de entrada que serão usados por ele. Por exemplo, quando chegar à fileira com a palavra-chave *Excluir_Usuario*, apenas o nome do usuário é obrigatório. Para acrescentar um novo usuário, o nome do usuário e a senha são dados obrigatórios. Os valores de entrada também podem ser extraído de um banco de dados, como demonstrado com a segunda palavra-chave *Adicionar_Usuario*, na qual a menção dos dados é digitada em vez dos próprios dados, o que gera maior flexibilidade na hora de acessar os dados que podem mudar com a execução dos testes. Isto permite que as técnicas orientadas a dados sejam combinadas com o esquema de palavras-chave.

Entre as questões a considerar estão:

- Quanto mais granulares forem as palavras, mais específicos serão os cenários que podem ser cobertos, mas a manutenção da linguagem de alto nível poderá ficar mais complexa;
- Permitir que o *Test Analyst* especifique ações de baixo nível (*ClicarBotao*, *SelecionarLista etc.*) deixa os testes com palavras-chave muito mais capazes de lidar com situações diferentes. No entanto, como estas ações estão ligadas diretamente à interface gráfica de usuário, também pode fazer que os testes exijam uma manutenção maior quando as mudanças forem realizadas;
- O uso de palavras-chave agregadas pode simplificar o desenvolvimento, mas complicar a manutenção. Por exemplo, podem existir seis palavras-chave diferentes que, coletivamente, criam um registro. Uma palavra-chave que realmente for composta por seis palavras-chave consecutivamente pode ser criada para simplificar a ação?
- Independentemente da análise da linguagem da palavra-chave, sempre haverá momentos em que palavras-chave novas e diferentes serão necessárias. Existem dois domínios diferentes em uma palavra-chave (isto é, a lógica de negócios por trás dela e a funcionalidade de automação para executá-la). Portanto, o processo deve ser criado para lidar com ambos os domínios.

A automação de testes baseados em palavras-chave pode reduzir bastante os custos de manutenção da automação de testes, mas é mais cara, mais difícil de desenvolver e leva mais tempo para projetar corretamente a fim de obter o retorno sobre o investimento esperado.

6.3 Ferramentas de teste específicas

Esta parte contém informações sobre as ferramentas que provavelmente serão utilizadas pelo *Technical Test Analyst* em situações que vão além das que aparecem no *Advanced Level Test Analyst Syllabus* [ISTQB_ALTA_SYL] e no *syllabus* do nível fundamental [ISTQB_FL_SYL].

6.3.1 Ferramentas de semeamento / injeção de falhas

As ferramentas de semeamento de falhas são utilizadas principalmente no código para a criação de tipos únicos ou limitados de falhas no código de maneira sistemática. Estas ferramentas inserem defeitos no objeto de teste propositalmente para avaliar a qualidade das suítes de teste (isto é, sua capacidade de detectar defeitos).

A injeção de falhas se concentra na realização de testes em qualquer mecanismo de tratamento de falhas incorporado ao objeto de teste, submetendo-o a condições anormais. As ferramentas de injeção de falhas fornecem ao *software* entradas incorretas propositalmente para garantir que o *software* consiga dar conta da falha.

Ambos os tipos de ferramentas são geralmente utilizados pelo *Technical Test Analyst*, mas também podem ser usadas pelo desenvolvedor na hora de testar códigos recém-desenvolvidos.

6.3.2 Ferramentas de teste de desempenho

As ferramentas de teste de desempenho possuem duas funções principais:

- Geração de carga;
- Medição e análise da resposta do sistema a determinada carga.

A geração de carga é realizada através da implementação de um perfil operacional predefinido (*vide* o item 4.5.4) como *script*. O *script* pode ser capturado inicialmente por um único usuário (talvez com a ferramenta de registro e execução) e, então, é implementado no perfil operacional específico com a ferramenta de teste de desempenho. A implementação deve levar em consideração a variação dos dados por transação (ou conjuntos de transações).

As ferramentas de desempenho geram uma carga através da simulação de grandes números de vários usuários (usuários "virtuais") de acordo com seus perfis operacionais designados para a geração de volumes específicos de dados de entrada. Em comparação com os *scripts* individuais de automação da execução de testes, muitos *scripts* de testes de desempenho reproduzem a interação do usuário com o sistema no nível do protocolo de comunicação e não mediante a simulação da interação do usuário através da interface gráfica de usuário. Isto geralmente reduz o número necessário de "sessões" diferentes durante os testes. Algumas ferramentas de geração de carga também podem levar o aplicativo, com sua interface de usuário, a uma medição mais atenta do tempo de resposta enquanto o sistema lida com a carga.

Várias medições são feitas pela ferramenta de teste de desempenho para possibilitar a análise durante ou após a execução do teste. Entre as medições feitas e os relatórios elaborados estão:

- O número de usuários simulados ao longo do teste;
- O número e o tipo de transações geradas pelos usuários simulados e o índice de chegada das transações;
- Os tempos de resposta a solicitações de transação específicas de usuários;
- A comparação de relatórios e gráficos de carga com os tempos de resposta;
- Os relatórios sobre a utilização de recursos (por exemplo, o uso no tempo com valores mínimos e máximos).

Entre os fatores importantes que é preciso levar em consideração na implementação das ferramentas de teste de desempenho estão:

- O *hardware* e a banda larga de rede necessários para a geração da carga;
- A compatibilidade da ferramenta com o protocolo de comunicação utilizado pelo sistema testado;
- A flexibilidade da ferramenta para permitir a fácil implementação de diferentes perfis operacionais;
- A necessidade de recursos de monitoramento, análise e divulgação.

As ferramentas de teste de desempenho são normalmente adquiridas e não desenvolvidas na empresa devido aos esforços necessários para desenvolvê-las. No entanto, talvez seja adequado desenvolver uma ferramenta de desempenho específica se as restrições técnicas impedirem a utilização de um produto disponível ou se o perfil de carga e os recursos fornecidos forem relativamente simples.

6.3.3 Ferramentas de teste baseado na rede

Várias ferramentas especializadas comerciais e de código aberto estão disponíveis para os testes na rede. A seguinte lista mostra a finalidade de algumas ferramentas comuns de testes baseados na rede:

- As ferramentas de teste de *hyperlink* são utilizadas para analisar e verificar a presença de *hyperlinks* truncados ou a ausência de *hyperlinks* em um *site*;
- Os verificadores de HTML e XML são ferramentas que verificam a conformidade das páginas criadas por um *site* com as normas HTML e XML;
- Os simuladores de carga servem para testar como o servidor reage quando grandes números de usuários estão conectados;
- As ferramentas leves de execução de automação que funcionam com navegadores diferentes;
- Ferramentas de varredura de servidores para a verificação de arquivos órfãos (sem links);
- Corretores ortográficos em HTML;
- Ferramentas de verificação de Cascading Style Sheet (CSS);
- Ferramentas para a verificação de infrações de normas, como a seção 508 da norma de acessibilidade nos EUA ou a M/376 na Europa;
- Ferramentas que detectam uma variedade de problemas de segurança.

Uma boa fonte de ferramentas de testes baseados na rede de código aberto é [Web-7]. A organização por trás deste *site* define as normas da Internet e fornece uma ampla variedade de ferramentas de verificação de erros com base nas normas.

Algumas ferramentas, entre as quais está um mecanismo de indexação automática que fornece informações sobre o tamanho das páginas e o tempo necessário para fazer o *download* delas e se a página existe ou não (por exemplo, erro HTTP 404). Ela fornece informações úteis ao desenvolver, ao *webmaster* e ao testador. O *Test Analyst* e o *Technical Test Analyst* utilizam estas ferramentas sobretudo durante os testes de sistema.

6.3.4 Ferramentas de suporte de testes baseados em modelos

Os testes baseados em modelos são uma técnica segundo a qual um modelo formal, como um autômato finito, é utilizado para descrever o comportamento pretendido do tempo de execução de um sistema controlado por *software*. As ferramentas comerciais de teste baseado em modelos (*vide* [Utting07]) frequentemente dispõem de um mecanismo que permite que o usuário "execute" o modelo. Cadeias de execução interessantes podem ser salvas e utilizadas como casos de teste. Outros modelos executáveis, como as redes de Petri e diagramas de transição de estados, também suportam os testes baseados em modelos. Os modelos (e as ferramentas) destes testes podem ser utilizados para a geração de grandes conjuntos de cadeias de execução diferentes.

As ferramentas de teste baseado em modelos podem facilitar a redução de um número muito grande de possíveis caminhos gerados em um modelo. Os testes com estas ferramentas representam um ponto de vista diferente do *software* testado. Isto pode levar à descoberta de defeitos que poderiam ter sido ignorados pelos testes funcionais.

6.3.5 Teste de componentes e ferramentas de automação de pacotes

Embora os testes de componentes e as ferramentas de automação de pacotes sejam ferramentas de desenvolvedores, em muitos casos, são utilizados e mantidos pelo *Technical Test Analyst*, especialmente em um contexto de desenvolvimento ágil.

Frequentemente, as ferramentas de teste de componentes são próprias da linguagem utilizada na programação de um módulo. Por exemplo, se a Java foi utilizada como linguagem de programação, a JUnit pode ser usada para automatizar os testes de unidade. Muitas outras linguagens possuem suas próprias ferramentas especiais de teste. Em conjunto, são chamadas de estrutura do xUnit. Tal estrutura gera objetos

de teste referentes a cada classe criada, simplificando, assim, as tarefas que o programador precisa realizar quando automatizar os testes de componente.

As ferramentas de depuração facilitam os testes de componentes manuais em um nível muito baixo, permitindo que o desenvolvedor e o *Technical Test Analyst* mudem valores variáveis durante a execução e repassem cada linha do código durante os testes. As ferramentas de depuração também são utilizadas para ajudar o desenvolver a isolar e identificar problemas no código quando uma falha for comunicada pela equipe de teste.

As ferramentas de automação de pacotes frequentemente permite o lançamento automático de novos pacotes sempre que um componente for alterado. Após a conclusão do pacote, outras ferramentas executam automaticamente os testes de componentes. Este nível de automação em torno ao processo de elaboração de pacotes é normalmente visto em um ambiente de integração contínua.

Quando configurado corretamente, este conjunto de ferramentas pode surtir um efeito muito positivo na qualidade dos pacotes lançados para teste. Se uma alteração feita por um programador introduzir defeitos de regressão no pacote, fará que alguns dos testes automatizados falhem, o que provocará uma investigação imediata da causa das falhas antes do lançamento do pacote no ambiente de teste.

7 Referências

7.1 Normas

As seguintes normas são mencionadas nestes capítulos:

- ANSI/IEEE Std 1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*. Capítulo 5;
- IEC-61508. Capítulo 2;
- [ISO25000] ISO/IEC 25000:2005, *Software Engineering – Software Product Quality Requirements and Evaluation* (SQuaRE). Capítulo 4;
- [ISO9126] ISO/IEC 9126-1:2001, *Software Engineering – Software Product Quality*. Capítulo 4;
- [RTCA DO-178B/ED-12B] *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/EUROCAE ED12B,1992. Capítulo 2.

7.2 Documentos da ISTQB

[ISTQB_AL_OVIEW] *ISTQB Advanced Level Overview*, versão 2012

[ISTQB_ALTA_SYL] *ISTQB Advanced Level Test Analyst Syllabus*, versão 2012 *ISTQB Foundation Level Syllabus*, versão 2011

[ISTQB_GLOSSARY] *ISTQB Glossary of Terms used in Software Testing*, versão 2.2, 2012

7.3 Livros

[Bass03] BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. *Software Architecture in Practice* – segunda edição, Addison-Wesley, 2003, ISBN 0-321-15495-9.

[Bath08] BATH, Graham; MCKAY, Judy. *The Software Test Engineer's Handbook* – Rocky Nook, 2008, ISBN 978-1-933952-24-6.

[Beizer90] BEIZER, Boris. *Software Testing Techniques* – segunda edição, International Thomson Computer Press, 1990, ISBN 1-8503-2880-3.

[Beizer95] BEIZER, Boris. *Black-Box Testing* – John Wiley & Sons, 1995, ISBN 0-471-12094-4.

[Buwalda01] BUWALDA, Hans. *Integrated Test Design and Automation* – Addison-Wesley Longman, 2001, ISBN 0-201-73725-6.

[Copeland03] COPELAND, Lee. *A Practitioner's Guide to Software Test Design* – Artech House, 2003, ISBN 1-58053-791-X.

[Gamma94] *Design Patterns*: *Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994, ISBN 0-201-63361-2.

[Jorgensen07] JORGENSEN, Paul C. *Software Testing, a Craftsman's Approach* – terceira edição, CRC Press, 2007, ISBN-13:978-0-8493-7475-3.

[Kaner02] BACH, James; KANER, Cem; PETTICHORD, Bret. *Lessons Learned in Software Testing* – Wiley, 2002, ISBN 0-471-08112-4;

[Koomen06] AALST, Leo van der; BROEKMAN, Bart; KOOMEN, Tim; VROON, Michiel. *TMap NEXT, For Result Driven Testing* – UTN Publishers, 2006, ISBN 90-72194-79-9.

[McCabe76] MCCABE, Thomas J. "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, dezembro de 1976, páginas 308-320.

[NIST96] MCCABE, Thomas J.; WATSON, Arthur H. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST edição especial 500-235, elaborada a pedido do NIST 43NANB517266, setembro de 1996.

[Splaine01] JASKIEL, Stefan P.; SPLAINE, Steven. *The Web-Testing Handbook* – STQE Publishing, 2001, ISBN 0-970-43630-0.

[Utting 07] LEGEARD, Bruno; UTTING, Mark. *Practical Model-Based Testing: A Tools Approach* – Morgan- Kaufmann, 2007, ISBN 978-0-12-372501-1.

[Whittaker04] THOMPSON, Herbert; WHITTAKER, James. *How to Break Software* Security – Pearson / Addison-Wesley, 2004, ISBN 0-321-19433-0.

[Wiegers02] WIEGERS, Karl. *Peer Reviews in Software: A Practical Guide* – Addison-Wesley, 2002, ISBN 0-201-73485-0.

7.4 Outras referências

As seguintes referências contêm informações disponíveis na Internet. Embora estas referências tenham sido acessadas até o momento da publicação deste *syllabus* de nível avançado, a ISTQB não pode ser responsabilizada se as referências não estiverem mais disponíveis.

[Web-1] www.testingstandards.co.uk

[Web-2] http://www.nist.gov – Instituto Nacional de Metrologia, Normalização e Qualidade Industrial dos EUA (NIST)

[Web-3] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx

[Web-4] http://portal.acm.org/citation.cfm?id=308798

[Web-5] http://www.processimpact.com/pr_goodies.shtml

[Web-6] http://www.ifsq.org

[Web-7] http://www.W3C.org

Capítulo 4: [Web-1], [Web-2]

Capítulo 5: [Web-3], [Web-4], [Web-5], [Web-6]

Capítulo 6: [Web-7]

8 Índice remissivo